

Mathematics of Neural Networks

Volume I: Foundations and Core Architectures

From first principles to the full backpropagation algorithm

Every derivation step by step

Every symbol defined

Explicit numerical examples

Geometric intuitions

Silvano Salvador

Contents

0.1	Number Sets	2
0.2	Logical and Set Notation	3
0.3	Vectors and Operations	4
0.4	Matrices and Multiplication	5
0.5	Differential Calculus	8
0.6	Essential Probability	9
0.7	Advanced Symbols Used Later	10
0.8	Matrix Calculus and the Jacobian	10
0.9	Eigenvalues, Eigenvectors, and the Hessian	11
0.10	Singular Value Decomposition (SVD)	12
0.11	Positive (Semi-)Definite Matrices	13
0.12	Information Theory Foundations	14
0.13	Convexity and the Optimization Landscape	15
1.1	The Mathematical Model	16
1.2	Activation Functions	17
1.3	Derivatives of Activation Functions — Step by Step	19
1.4	The GELU Activation Function	21
1.5	Swish and SiLU	21
1.6	Universal Approximation Theorem	22
1.7	Activation Function Summary and Selection Guide	23
2.1	Architecture and Complete Notation	25
2.2	Forward Propagation	26
2.3	Complete Numerical Example — Forward Pass	26
2.4	Vectorised Batch Forward Pass	27
2.5	Depth, Width, and Representational Power	28
2.6	Forward Mode vs. Reverse Mode Automatic Differentiation	29
2.7	Memory Complexity of Forward and Backward Passes	29

2.8	Computational Graph and Dynamic vs. Static Computation	30
2.9	Parameter Sharing, Equivariance, and Invariance	30
3.1	Mean Squared Error (MSE)	32
3.2	Binary Cross-Entropy (BCE)	33
3.3	Softmax and Categorical Cross-Entropy	34
3.4	Huber Loss and Robust Regression	35
3.5	Label Smoothing	36
3.6	Focal Loss for Class Imbalance	36
3.7	Surrogate Losses and Zero-One Loss	37
3.8	Numerical Example: Loss Surfaces and Saddle Points	38
3.9	Contrastive and Ranking Losses	38
4.1	The Chain Rule on Computational Graphs	40
4.2	Complete Algorithm	41
4.3	Complete Numerical Example — Backpropagation	42
4.4	Vector-Jacobian Products and Efficient Backpropagation	43
4.5	Backpropagation Through Non-Smooth Operations	43
5.1	Gradient Descent and Variants	45
5.2	Adam — Every Step Explained	46
5.3	AdamW and Decoupled Weight Decay	47
5.4	Convergence Analysis	48
5.5	Learning Rate Warmup and Cosine Annealing	50
5.6	Gradient Clipping and Training Stability	50
5.7	Second-Order Optimisation Methods	51
6.1	L1 and L2 Regularization	53
6.2	Dropout	54
6.3	Batch Normalization — Every Step	55
6.4	Layer Normalisation vs. Batch Normalisation	56
6.5	Stochastic Depth	56

6.6	Why Dropout Works: Ensemble Interpretation	57
6.7	Batch Normalisation: Full Derivation of Backpropagation	58
6.8	Data Augmentation as Regularisation	58
7.1	The Discrete Convolution — Every Step	60
7.2	Pooling and Its Backpropagation	61
7.3	Residual Connections — Why They Solve the Degradation Problem	62
7.4	Transposed Convolution	63
7.5	Depthwise Separable Convolutions	63
7.6	Backpropagation Through a Residual Block	64
7.7	FLOPs Analysis: Standard vs. Depthwise Separable Convolution	65
8.1	RNN Base	66
8.2	BPTT and Vanishing/Exploding Gradients — Derivation	66
8.3	LSTM — Every Gate Explained	67
8.4	Gated Recurrent Unit (GRU)	68
8.5	Bidirectional and Stacked RNNs	69
8.6	LSTM Backpropagation Through Time — Gradient Analysis	69
8.7	GRU vs. LSTM: Parameter Count and Speed	70
9.1	The Attention Intuition	71
9.2	Self-Attention — Complete Derivation	71
9.3	Multi-Head Attention — Why Multiple Heads	73
9.4	Positional Encoding	74
9.5	The Complete Transformer Block	75
9.6	Efficient Attention Variants	76
9.7	Relative and ALiBi Positional Encoding	77
9.8	Flash Attention: IO-Aware Self-Attention	77
9.9	Rotary Position Embeddings (RoPE)	78
10.1	Encoder–Decoder Architecture	79
10.2	Cross-Attention	80

10.3	Complete Decoder Block	81
10.4	Generation Strategies	81
10.5	Sampling Strategies for Text Generation	82
10.6	Encoder–Decoder vs. Decoder-Only Architecture	83
11.1	Weight Initialisation	84
11.2	Learning Rate Scheduling	85
11.3	Complete Training Loop	86
11.4	Mixed Precision Training	87
11.5	Gradient Checkpointing	88
11.6	ZeRO: Zero Redundancy Optimizer	89
11.7	Pipeline and Tensor Parallelism	89
12.1	Problem Setup	90
12.2	Forward Pass on Full Batch	90
12.3	Gradient Descent Steps	91
12.4	Summary of Volume I	92
12.5	XOR Convergence — Extended Analysis	92
12.6	Key Theorems and Results in Volume I	93
12.7	Architecture Design Checklist	94
12.8	Architecture Design Checklist	94
12.9	Summary and Connections to Volume II	96
	Dimostrazioni Differite	97
	Ricettario di Derivate per le Reti Neurali	98
	Complessità Computazionale e di Memoria	99
	Glossario dei Simboli	99
	Bibliografia	100

Notation, Symbols and Prerequisites

Every mathematical symbol used in this volume is defined here, including those introduced in later parts. No prerequisite beyond high-school algebra is assumed.

0.1 Number Sets

The mathematics of neural networks rests on a precise hierarchy of number sets, each with distinct algebraic and topological properties. Understanding this hierarchy is fundamental for correctly interpreting the domain and codomain of every operation encountered in the following chapters.

Hierarchy of Number Sets

- $\mathbb{N} = \{0, 1, 2, 3, \dots\}$: the **natural numbers**, used to count neurons, layers, epochs, and batch indices.
- $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$: the **integers**, used for strides and padding in convolutions.
- $\mathbb{Q} = \{p/q \mid p, q \in \mathbb{Z}, q \neq 0\}$: the **rationals**, dense in $\mathbb{R}$ but lacking completeness.
- $\mathbb{R}$: the **reals**, the space in which weights, activations, and gradients live.
- $\mathbb{C} = \{a + bi \mid a, b \in \mathbb{R}\}$: the **complex numbers**, useful in the DFT and spectral kernels.

The inclusion chain is $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$.

Why the reals? The real numbers are the unique Dedekind-*complete* ordered field: every non-empty set that is bounded above has a least upper bound. This property guarantees that Cauchy sequences — such as those generated by gradient updates — actually converge to a limit within $\mathbb{R}$.

Floating point and approximation. In practice, a computer represents numbers in floating point according to the IEEE 754 standard. A `float32` uses 1 sign bit, 8 exponent bits, and 23 mantissa bits, covering approximately $[1.2 \times 10^{-38}, 3.4 \times 10^{38}]$ with about 7 significant digits. The `bfloat16` format, widely used in LLM training, sacrifices mantissa precision (7 bits) to preserve the exponent range (8 bits), halving memory with minimal impact on optimisation.

EXAMPLE — Numbers and Tensor Shapes

In PyTorch, a network with $n = 512$ hidden neurons, batch $B = 32$, sequence $T = 128$, embedding dimension $d = 768$:

$$B \times T \times d = 3,145,728 \text{ real elements per batch}$$

$$\boxed{32 \times 128 \times 768}$$

Each element is a `float32` (4 bytes), so ≈ 12 MB per activation of a single layer.

Frequently used intervals and sets.

- $[a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$: closed interval.
- $(0, 1)$: probabilities, outputs of sigmoid and softmax.

- $\mathbb{R}^+ = (0, +\infty)$: learning rates, variances, normalisation scales.
- $\mathbb{R}^n$: space of activation vectors.
- $\mathbb{R}^{m \times n}$: space of weight matrices.

The distinction between $\mathbb{R}$ as a continuous space and its floating-point representation is crucial for *numerical stability*. Operations such as $\log(1 + e^x)$ (softplus) or $\log \sum_i e^{x_i}$ (logsumexp) must be implemented using the *log-sum-exp trick* to avoid overflow when $x \gg 0$.

0.2 Logical and Set Notation

Set-theoretic and logical notation is the language with which architectures, loss functions, and hypothesis spaces are defined precisely and unambiguously.

Fundamental Set Operators

Given two sets A and B in a universal space Ω :

- **Union:** $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
- **Intersection:** $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$
- **Difference:** $A \setminus B = \{x \mid x \in A, x \notin B\}$
- **Complement:** $A^c = \Omega \setminus A$
- **Cartesian product:** $A \times B = \{(a, b) \mid a \in A, b \in B\}$
- **Power:** $A^n = A \times A \times \dots \times A$ (n times)

Logical quantifiers. Two quantifiers structure every mathematical statement:

$$\forall x \in A, P(x) \quad (\text{"for every } x \text{ in } A, P(x) \text{ holds"})$$

$$\exists x \in A \text{ s.t. } P(x) \quad (\text{"there exists at least one } x \text{ in } A \text{ for which } P(x) \text{ holds"})$$

Negation swaps the quantifiers: $\neg(\forall x, P(x)) \Leftrightarrow \exists x, \neg P(x)$.

Implication and biconditional.

- $P \Rightarrow Q$: " P implies Q ". Equivalent to $\neg P \vee Q$.
- $P \Leftrightarrow Q$: " P if and only if Q ". True when P and Q have the same truth value.

EXAMPLE — Notation in the Loss Function

Let $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$ be the training dataset. The correct-classification condition for a classifier f_θ is:

$$\forall i \in \{1, \dots, N\}, \quad \arg \max_k f_\theta(\mathbf{x}_i)_k = y_i$$

Empirical risk minimisation seeks:

$$\theta^* = \arg \min_{\theta \in \Theta} \frac{1}{N} \sum_{i=1}^N \ell(f_\theta(\mathbf{x}_i), y_i)$$

where $\Theta \subset \mathbb{R}^P$ is the parameter space with P the total number of weights.

Set-builder notation. The form $\{f(x) \mid P(x)\}$ (“the set of all $f(x)$ such that $P(x)$ ”) is ubiquitous:

$$\mathcal{B}(\mathbf{x}, r) = \{\mathbf{y} \in \mathbb{R}^n \mid \|\mathbf{y} - \mathbf{x}\|_2 < r\} \quad (\text{open ball})$$

$$\Delta^{K-1} = \left\{ \mathbf{p} \in \mathbb{R}^K \mid p_k \geq 0, \sum_{k=1}^K p_k = 1 \right\} \quad (\text{probability simplex})$$

Cardinality and Infinite Sets

- $|A|$: cardinality (number of elements) of A .
- $|\mathbb{N}| = \aleph_0$: countable cardinality.
- $|\mathbb{R}| = \mathfrak{c} = 2^{\aleph_0}$: cardinality of the continuum; uncountable.

In ML, datasets have finite cardinality N , but the space of functions approximated by a network is infinite-dimensional.

Functions as relations. A function $f : A \rightarrow B$ is a relation $f \subset A \times B$ such that for every $a \in A$ there exists *exactly* one $b \in B$ with $(a, b) \in f$. The key structural properties are:

- **Injective:** $f(a_1) = f(a_2) \Rightarrow a_1 = a_2$
- **Surjective:** $\forall b \in B, \exists a \in A$ s.t. $f(a) = b$
- **Bijective:** injective and surjective; f^{-1} exists and is a function

0.3 Vectors and Operations

Vectors are the fundamental objects of deep learning: every sample, every activation, every gradient is a vector. Mastering their algebra is an absolute prerequisite.

Vector in $\mathbb{R}^n$

A column vector $\mathbf{x} \in \mathbb{R}^n$ is an ordered sequence of n real numbers:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad x_i \in \mathbb{R}$$

The *transpose* $\mathbf{x}^\top \in \mathbb{R}^{1 \times n}$ is a row vector.

Elementary operations.

Sum: $(\mathbf{x} + \mathbf{y})_i = x_i + y_i$. Geometrically, appends the arrow of $\mathbf{y}$ to the tip of $\mathbf{x}$ (parallelogram rule).

Scaling: $(\alpha \mathbf{x})_i = \alpha x_i$. Stretches or compresses the vector; if $\alpha < 0$, reverses it.

Dot product:

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^\top \mathbf{y} = \sum_{i=1}^n x_i y_i = \|\mathbf{x}\|_2 \|\mathbf{y}\|_2 \cos \theta$$

where θ is the angle between the two vectors. The dot product measures *directional similarity*: it is zero for orthogonal vectors and maximum for parallel vectors.

EXAMPLE — Dot Product in a Linear Layer

In a single perceptron with $n = 3$ inputs, the pre-activation z is:

$$z = \mathbf{w}^\top \mathbf{x} + b = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

With $\mathbf{w} = (0.5, -1.2, 0.8)^\top$ and $\mathbf{x} = (1.0, 0.5, 2.0)^\top$, $b = 0.1$:

$$z = 0.5 \cdot 1.0 + (-1.2) \cdot 0.5 + 0.8 \cdot 2.0 + 0.1 = 0.5 - 0.6 + 1.6 + 0.1 = 1.6$$

Vector norms. A norm measures the “length” of a vector:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i| \quad (L^1 \text{ norm, promotes sparsity})$$

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2} \quad (\text{Euclidean norm})$$

$$\|\mathbf{x}\|_\infty = \max_i |x_i| \quad (\text{max norm})$$

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^n |x_i|^p \right)^{1/p} \quad (L^p \text{ norm, } p \geq 1)$$

Outer product. While the dot product $\mathbf{x}^\top \mathbf{y}$ produces a scalar, the outer product $\mathbf{x} \mathbf{y}^\top$ produces a rank-1 matrix:

$$(\mathbf{x} \mathbf{y}^\top)_{ij} = x_i y_j \in \mathbb{R}^{m \times n}, \quad \mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n$$

It appears in the Adam update (first and second moments), in the gradient computation of a linear layer, and in the SVD factorisation.

Orthogonality and Orthonormality

- $\mathbf{x} \perp \mathbf{y}$ (orthogonal) $\Leftrightarrow \mathbf{x}^\top \mathbf{y} = 0$.
- A set $\{\mathbf{e}_1, \dots, \mathbf{e}_k\}$ is **orthonormal** if $\mathbf{e}_i^\top \mathbf{e}_j = \delta_{ij}$ (Kronecker delta).
- The canonical basis of $\mathbb{R}^n$: $\mathbf{e}_i$ has 1 in position i and 0 elsewhere.

Projection. The projection of $\mathbf{x}$ onto $\mathbf{u}$ (with $\|\mathbf{u}\|_2 = 1$) is:

$$\text{proj}_{\mathbf{u}} \mathbf{x} = (\mathbf{x}^\top \mathbf{u}) \mathbf{u}$$

This is the building block of the Gram-Schmidt algorithm and principal component analysis (PCA).

Broadcasting in NumPy/PyTorch. Operations between vectors of different shapes (e.g., $\mathbf{x} \in \mathbb{R}^n$ and a scalar α) are handled automatically by broadcasting. However, it is essential to distinguish a column vector $(n, 1)$ from a row vector $(1, n)$ to avoid silent bugs in matrix-vector computations.

0.4 Matrices and Multiplication

A matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is a rectangular array of real numbers. Matrices are the fundamental data structure of neural networks: weights, activations (as batches), Jacobians, Gram matrices, and attention scores are all matrices.

Basic operations.

- **Addition:** $(A + B)_{ij} = A_{ij} + B_{ij}$. Requires $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$ (same shape).
- **Scalar multiplication:** $(\alpha \mathbf{A})_{ij} = \alpha A_{ij}$.
- **Matrix multiplication:** $(\mathbf{AB})_{ij} = \sum_{k=1}^p A_{ik} B_{kj}$ for $\mathbf{A} \in \mathbb{R}^{m \times p}$, $\mathbf{B} \in \mathbb{R}^{p \times n}$. Result $\in \mathbb{R}^{m \times n}$. Cost: $O(mnp)$ multiplications.
- **Transpose:** $(\mathbf{A}^\top)_{ij} = A_{ji}$. Properties: $(\mathbf{AB})^\top = \mathbf{B}^\top \mathbf{A}^\top$, $((\mathbf{A}^\top)^\top) = \mathbf{A}$.

Trace and Frobenius norm.

$$\text{tr}(\mathbf{A}) = \sum_{i=1}^n A_{ii} \quad (\text{for square } \mathbf{A}), \quad \|\mathbf{A}\|_F = \sqrt{\sum_{i,j} A_{ij}^2} = \sqrt{\text{tr}(\mathbf{A}^\top \mathbf{A})}$$

Key trace identity: $\text{tr}(\mathbf{AB}) = \text{tr}(\mathbf{BA})$ (cyclic property). Used in deriving the gradient of $\text{tr}(\mathbf{W}^\top \mathbf{W}) = \|\mathbf{W}\|_F^2$: $\partial \|\mathbf{W}\|_F^2 / \partial \mathbf{W} = 2\mathbf{W}$ (L2 regularisation gradient).

Determinant. For square $\mathbf{A} \in \mathbb{R}^{n \times n}$:

- 2×2 : $\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$.
- General: expansion by cofactors or LU decomposition.
- Geometric meaning: $|\det \mathbf{A}|$ is the volume scaling factor of the linear map. $\det \mathbf{A} = 0$ iff $\mathbf{A}$ is singular (non-invertible).
- Product rule: $\det(\mathbf{AB}) = \det \mathbf{A} \cdot \det \mathbf{B}$.

EXAMPLE — Matrix Multiplication in a Neural Network Layer

Batch $\mathbf{X} \in \mathbb{R}^{4 \times 3}$ (4 examples, 3 features), weight $\mathbf{W} \in \mathbb{R}^{3 \times 2}$, bias $\mathbf{b} \in \mathbb{R}^2$. Pre-activation:

$$\mathbf{Z} = \mathbf{XW} + \mathbf{1b}^\top \in \mathbb{R}^{4 \times 2}$$

Broadcasting: $\mathbf{1} \in \mathbb{R}^4$ (column of ones). Cost: $4 \times 3 \times 2 = 24$ multiplications for $\mathbf{XW}$, plus $4 \times 2 = 8$ additions for bias. On a GPU, the matrix product is accelerated by Tensor Cores to ~ 312 TFLOPS (A100), processing all 24 operations in parallel.

Matrices are the means by which neural networks apply linear transformations. Every fully-connected layer is a matrix multiplication; every convolution is a matrix multiplication in Toeplitz or im2col form.

Matrix and Basic Operations

A matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ has m rows and n columns, with the (i, j) -th element denoted A_{ij} or $[\mathbf{A}]_{ij}$.

- **Transposition:** $[\mathbf{A}^\top]_{ij} = A_{ji}$; shape: $\mathbb{R}^{n \times m}$.
- **Symmetry:** $\mathbf{A} = \mathbf{A}^\top$ (necessarily square).
- **Identity:** $\mathbf{I}_n$ with $[\mathbf{I}]_{ij} = \delta_{ij}$.
- **Diagonal:** $\mathbf{D} = \text{diag}(d_1, \dots, d_n)$ with $D_{ij} = d_i \delta_{ij}$.

Matrix product. Given $\mathbf{A} \in \mathbb{R}^{m \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times n}$:

$$[\mathbf{AB}]_{ij} = \sum_{l=1}^k A_{il} B_{lj}$$

The product is defined only when the “inner” dimensions match ($k = k$). The result is in $\mathbb{R}^{m \times n}$.

Geometric interpretations of matrix multiplication.

As dot products of rows and columns. $[\mathbf{AB}]_{ij}$ is the dot product between row i of $\mathbf{A}$ and column j of $\mathbf{B}$.

As a linear combination of columns. $\mathbf{Ax} = x_1 \mathbf{a}_1 + x_2 \mathbf{a}_2 + \cdots + x_n \mathbf{a}_n$, where $\mathbf{a}_j$ are the columns of $\mathbf{A}$. The image of $\mathbf{A}$ is the *span* of its columns.

EXAMPLE — Multiplication in a Fully-Connected Layer

Layer with $n = 4$ inputs and $m = 3$ outputs, batch $B = 2$:

$$\mathbf{W} = \begin{pmatrix} 0.1 & -0.2 & 0.3 & 0.0 \\ 0.5 & 0.1 & -0.4 & 0.2 \\ -0.1 & 0.3 & 0.2 & -0.3 \end{pmatrix} \in \mathbb{R}^{3 \times 4}$$

$$\mathbf{X} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 2 \\ 0.5 & -0.5 \end{pmatrix} \in \mathbb{R}^{4 \times 2}$$

$$\mathbf{Z} = \mathbf{WX} + \mathbf{b}\mathbf{1}^\top \in \mathbb{R}^{3 \times 2}$$

Computing $\mathbf{WX}$ requires $2 \times 3 \times 4 = 24$ multiplications and 21 additions (in general $2mnk - mk$ FLOPs).

Algebraic properties.

- $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$ (associativity)
- $\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$ (distributivity)
- $(\mathbf{AB})^\top = \mathbf{B}^\top \mathbf{A}^\top$ (transpose of a product)
- In general $\mathbf{AB} \neq \mathbf{BA}$ (**non-commutative!**)

Trace and determinant.

$$\text{tr}(\mathbf{A}) = \sum_{i=1}^n A_{ii}, \quad \text{tr}(\mathbf{AB}) = \text{tr}(\mathbf{BA})$$

The determinant $\det(\mathbf{A})$ measures the volume scaling factor of the linear transformation; $\det(\mathbf{A}) = 0 \Leftrightarrow \mathbf{A}$ is singular (non-invertible).

Inverse and Pseudo-inverse

- If $\mathbf{A} \in \mathbb{R}^{n \times n}$ is invertible: $\mathbf{A}^{-1} \mathbf{A} = \mathbf{A} \mathbf{A}^{-1} = \mathbf{I}$.
- **Moore-Penrose pseudo-inverse** $\mathbf{A}^+$: solution to the over/under-determined system in the least-squares sense. If $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ is the SVD, then $\mathbf{A}^+ = \mathbf{V}\mathbf{\Sigma}^+ \mathbf{U}^\top$ where $\Sigma_{ii}^+ = 1/\sigma_i$ if $\sigma_i > 0$, else 0.

Computational complexity. The naïve product $\mathbf{A}(m \times k) \cdot \mathbf{B}(k \times n)$ is $O(mkn)$. For $n \times n$ matrices, Strassen achieves $O(n^{2.807})$; modern algorithms reach $O(n^{2.37})$. In practice, GPUs use BLAS (cuBLAS) with tiling optimised for the memory hierarchy.

Einsum notation. In PyTorch/NumPy, `torch.einsum('ij,jk->ik', A, B)` expresses matrix multiplication in explicit index notation, allowing arbitrary tensor contractions with a single optimised call.

0.5 Differential Calculus

Derivative. For $f : \mathbb{R} \rightarrow \mathbb{R}$, the derivative at x is:

$$f'(x) = \frac{df}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

Chain rule. If $g = f \circ u$, i.e. $g(x) = f(u(x))$:

$$\frac{dg}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$$

The chain rule is the mathematical foundation of backpropagation (Part 4): a composed network $f = f_L \circ \dots \circ f_1$ has gradient $\frac{\partial f}{\partial x} = \prod_{l=L}^1 \frac{\partial f_l}{\partial f_{l-1}}$.

Partial derivative. For $f : \mathbb{R}^n \rightarrow \mathbb{R}$, the partial derivative with respect to x_i treats all other variables as constants:

$$\frac{\partial f}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(\dots, x_i + h, \dots) - f(\dots, x_i, \dots)}{h}$$

Gradient. The gradient of $f : \mathbb{R}^n \rightarrow \mathbb{R}$ collects all partial derivatives:

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix} \in \mathbb{R}^n$$

The gradient always points in the direction of steepest ascent of f . Moving opposite to the gradient (gradient descent) decreases f : $f(\mathbf{x} - \eta \nabla f(\mathbf{x})) < f(\mathbf{x})$ for sufficiently small $\eta > 0$, provided $\nabla f(\mathbf{x}) \neq \mathbf{0}$.

Taylor expansion. The first-order Taylor approximation of f near $\mathbf{x}_0$:

$$f(\mathbf{x}_0 + \square) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^\top \square$$

Second-order (including curvature via the Hessian $\mathbf{H}$):

$$f(\mathbf{x}_0 + \square) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^\top \square + \frac{1}{2} \square^\top \mathbf{H}(\mathbf{x}_0) \square$$

where $H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$. The second-order term determines whether $\mathbf{x}_0$ is a minimum ($\mathbf{H} \succ 0$), maximum ($\mathbf{H} \prec 0$), or saddle point ($\mathbf{H}$ indefinite).

Critical points. A point $\mathbf{x}^*$ is critical if $\nabla f(\mathbf{x}^*) = \mathbf{0}$. For neural network loss surfaces, saddle points are far more common than local minima in high dimensions (Dauphin et al., 2014) — gradient descent escapes them via stochastic noise.

0.6 Essential Probability

Random variable. A variable X whose value is determined by a random experiment. Discrete X takes countably many values; continuous X takes values in $\mathbb{R}$ with a probability density function (pdf) $p(x) \geq 0$, $\int p(x) dx = 1$.

Expected value. For a discrete X taking values $x_1, \dots, x_k$:

$$\mathbb{E}[X] = \sum_{i=1}^k x_i P(X = x_i)$$

For continuous X : $\mathbb{E}[X] = \int x p(x) dx$. Linearity: $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$.

Variance and standard deviation.

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2, \quad \text{SD}(X) = \sqrt{\text{Var}(X)}$$

Gaussian distribution. $X \sim \mathcal{N}(\mu, \sigma^2)$ has pdf:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)$$

$\mathbb{E}[X] = \mu$, $\text{Var}(X) = \sigma^2$. Multivariate: $\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ with pdf $p(\mathbf{x}) \propto \exp(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \Sigma^{-1}(\mathbf{x} - \boldsymbol{\mu}))$. The Gaussian is the unique maximum-entropy distribution with fixed mean and variance — hence its ubiquity in weight initialisation and noise models.

Probability distribution over classes. For K mutually exclusive outcomes:

$$P(y = k), \quad k = 1, \dots, K, \quad \sum_{k=1}^K P(y = k) = 1$$

Bayes' theorem. Relates prior $P(\theta)$, likelihood $P(D|\theta)$, and posterior $P(\theta|D)$:

$$P(\theta|D) = \frac{P(D|\theta) P(\theta)}{P(D)}$$

Neural network training by maximum likelihood estimation (MLE) maximises $\log P(D|\theta)$ over parameters θ , which is equivalent to minimising the negative log-likelihood loss. Cross-entropy loss $= -\log P(y|\mathbf{x}; \theta)$ is the per-sample negative log-likelihood.

Independence. X and Y are independent if $P(X, Y) = P(X)P(Y)$. The i.i.d. (independent and identically distributed) assumption underlies the factorisation of the dataset likelihood:

$$P(D|\theta) = \prod_{i=1}^N P(y_i|\mathbf{x}_i; \theta)$$

which turns the product into a sum when taking logarithms — yielding the additive loss over training examples.

0.7 Advanced Symbols Used Later

Symbol	Meaning
$\mathbf{W}^{(l)}$	Weight matrix of layer l
$\mathbf{b}^{(l)}$	Bias vector of layer l
$\mathbf{z}^{(l)}$	Pre-activation vector of layer l
$\mathbf{a}^{(l)}$	Post-activation (output) vector of layer l
$\mathbf{x}^{(0)} = \mathbf{x}$	Network input
$\hat{\mathbf{y}}$	Network output (prediction)
$\mathbf{y}$	True target
$\mathcal{L}$	Loss (cost) function
$\delta^{(l)}$	Error signal (gradient) at layer l
$\odot$	Element-wise (Hadamard) product
σ	Sigmoid activation function
ϕ	Generic activation function
L	Total number of layers
n_l	Number of neurons in layer l
N	Number of training examples
α	Learning rate

0.8 Matrix Calculus and the Jacobian

In neural networks the loss $\mathcal{L}$ is a scalar function of vector-valued intermediate quantities. We need to differentiate scalars with respect to vectors, and vectors with respect to vectors.

Definition: Gradient and Jacobian

- **Gradient.** If $f : \mathbb{R}^n \rightarrow \mathbb{R}$, then $\nabla_{\mathbf{x}} f \in \mathbb{R}^n$ with $(\nabla f)_i = \partial f / \partial x_i$.
- **Jacobian.** If $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$, then $J \in \mathbb{R}^{m \times n}$ with $J_{ij} = \partial f_i / \partial x_j$.

Chain rule in matrix form. If $\mathbf{h} = \mathbf{f}(\mathbf{g}(\mathbf{x}))$, then:

$$\frac{\partial \mathbf{h}}{\partial \mathbf{x}} = \frac{\partial \mathbf{f}}{\partial \mathbf{g}} \cdot \frac{\partial \mathbf{g}}{\partial \mathbf{x}} \quad (\text{Jacobians multiply})$$

For a scalar loss $\mathcal{L} = f(\mathbf{g}(\mathbf{x}))$:

$$\nabla_{\mathbf{x}} \mathcal{L} = \left(\frac{\partial \mathbf{g}}{\partial \mathbf{x}} \right)^{\top} \nabla_{\mathbf{g}} \mathcal{L}$$

Important identities. For $\mathbf{y} = \mathbf{A}\mathbf{x}$ (with $\mathbf{A}$ constant):

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{A} \quad (\text{Jacobian}), \quad \frac{\partial (\mathbf{c}^{\top} \mathbf{y})}{\partial \mathbf{x}} = \mathbf{A}^{\top} \mathbf{c}$$

EXAMPLE — Jacobian of affine layer

Let $\mathbf{z} = \mathbf{W}\mathbf{a} + \mathbf{b}$ with $\mathbf{W} \in \mathbb{R}^{m \times n}$, $\mathbf{a} \in \mathbb{R}^n$. Then:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{a}} = \mathbf{W} \in \mathbb{R}^{m \times n}$$

So if $\mathcal{L}$ is a scalar and $\boldsymbol{\square} = \nabla_{\mathbf{z}} \mathcal{L} \in \mathbb{R}^m$:

$$\nabla_{\mathbf{a}} \mathcal{L} = \mathbf{W}^{\top} \boldsymbol{\square} \in \mathbb{R}^n$$

This is exactly the gradient propagation step in backpropagation (Part 4).

0.9 Eigenvalues, Eigenvectors, and the Hessian

Eigenvalues and eigenvectors reveal the intrinsic geometry of linear transformations. They are central to understanding gradient descent dynamics, weight matrix conditioning, and principal component analysis.

Eigenvalue Equation

A non-zero vector $\mathbf{v} \in \mathbb{R}^n$ is an *eigenvector* of $\mathbf{A} \in \mathbb{R}^{n \times n}$ with *eigenvalue* $\lambda \in \mathbb{C}$ if:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

Geometrically: $\mathbf{A}$ scales $\mathbf{v}$ by λ without rotating it (if $\lambda \in \mathbb{R}$). Computing: eigenvalues are the roots of the *characteristic polynomial* $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$.

Spectral norm. The *spectral norm* (operator 2-norm) of $\mathbf{A}$ is:

$$\|\mathbf{A}\|_2 = \sigma_{\max}(\mathbf{A}) = \sqrt{\lambda_{\max}(\mathbf{A}^\top \mathbf{A})}$$

For symmetric PSD matrices: $\|\mathbf{A}\|_2 = \lambda_{\max}(\mathbf{A})$. Spectral normalisation constrains $\|\mathbf{W}\|_2 \leq 1$ in GAN training (Miyato et al., 2018) for Lipschitz-stability of the discriminator.

Gradient descent and eigenvalues. On a quadratic $\mathcal{L} = \frac{1}{2}\mathbf{x}^\top \mathbf{H}\mathbf{x}$, gradient descent in the eigen-basis $\{\mathbf{v}_i\}$ decomposes into n independent 1D problems:

$$\theta_i^{(t+1)} = (1 - \eta\lambda_i)\theta_i^{(t)}$$

Convergence condition: $|1 - \eta\lambda_i| < 1$ for all i . This requires $0 < \eta < 2/\lambda_{\max}$. The convergence rate for each direction i is $|1 - \eta\lambda_i|$. The slowest-converging direction (worst case) has rate $\max_i |1 - \eta\lambda_i|$.

With the optimal step size $\eta = 2/(\lambda_{\min} + \lambda_{\max})$:

$$\rho = \frac{\kappa - 1}{\kappa + 1}, \quad \kappa = \lambda_{\max}/\lambda_{\min}$$

Large condition number κ causes slow convergence: the loss decreases slowly along the direction corresponding to $\lambda_{\min}$ (flattest curvature). Adam addresses this by adapting step sizes per-coordinate, effectively preconditioned by the empirical condition number of each parameter.

EXAMPLE — Eigenanalysis of a 2D Gradient Descent

$\mathcal{L}(\theta_1, \theta_2) = 4\theta_1^2 + 0.04\theta_2^2$. Hessian: $\mathbf{H} = \text{diag}(8, 0.08)$. Eigenvalues: $\lambda_1 = 8$, $\lambda_2 = 0.08$, $\kappa = 100$.

Optimal step size: $\eta^* = 2/(8 + 0.08) \approx 0.248$.

Convergence rates: in θ_1 direction: $|1 - 0.248 \times 8| = |-0.984| = 0.984$; in θ_2 direction: $|1 - 0.248 \times 0.08| = 0.980$.

Both rates ≈ 0.98 — the optimal step balances the two extremes. Steps to halve the error: $\log 0.5 / \log 0.98 \approx 34$. Compare: gradient descent on λ_1 alone with $\eta = 1/8$: converges in $\log 0.5 / \log 0 = 1$ step. Ill-conditioning forces a conservative step for the whole system.

Eigendecomposition. For a square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, a non-zero vector $\mathbf{v}$ and scalar λ satisfying $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ are called an *eigenvector* and *eigenvalue*, respectively.

Symmetric matrices. If $\mathbf{A} = \mathbf{A}^\top$, all eigenvalues are real and eigenvectors corresponding to distinct eigenvalues are orthogonal. The spectral theorem guarantees $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top$ where $\mathbf{Q}$ is orthogonal and $\mathbf{\Lambda} = \text{diag}(\lambda_1, \dots, \lambda_n)$.

Positive definite matrices. $\mathbf{A} \succ 0$ iff all eigenvalues are positive, iff $\mathbf{x}^\top \mathbf{A} \mathbf{x} > 0$ for all $\mathbf{x} \neq \mathbf{0}$.

Definition: Hessian Matrix

For $f : \mathbb{R}^n \rightarrow \mathbb{R}$ twice differentiable, the Hessian $\mathbf{H}(f) \in \mathbb{R}^{n \times n}$ is:

$$H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

At a critical point $\mathbf{x}^*$ (where $\nabla f = \mathbf{0}$):

- $\mathbf{H} \succ 0$ (all eigenvalues > 0): local minimum.
- $\mathbf{H} \prec 0$ (all eigenvalues < 0): local maximum.
- Mixed eigenvalues: saddle point.

The Hessian appears in second-order optimisation methods (natural gradient, K-FAC — Volume IV) and in the analysis of loss landscape geometry.

0.10 Singular Value Decomposition (SVD)

Every matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ admits the SVD:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$$

where $\mathbf{U} \in \mathbb{R}^{m \times m}$ and $\mathbf{V} \in \mathbb{R}^{n \times n}$ are orthogonal, and $\mathbf{\Sigma} \in \mathbb{R}^{m \times n}$ is diagonal with non-negative entries $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_{\min(m,n)} \geq 0$ (the *singular values*).

Geometric interpretation. The SVD decomposes any linear map $\mathbf{A} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ into three steps: (1) rotate the input by $\mathbf{V}^\top$ (orthogonal), (2) scale each axis by σ_i (possibly to zero), (3) rotate the result by $\mathbf{U}$ (orthogonal). The column vectors $\mathbf{u}_i$ of $\mathbf{U}$ are the *left singular vectors* (output directions); $\mathbf{v}_i$ of $\mathbf{V}$ are *right singular vectors* (input directions).

Eckart–Young theorem (best low-rank approximation). The best rank- r approximation to $\mathbf{A}$ in Frobenius and spectral norm is:

$$\mathbf{A}_r = \sum_{i=1}^r \sigma_i \mathbf{u}_i \mathbf{v}_i^\top = \mathbf{U}_r \mathbf{\Sigma}_r \mathbf{V}_r^\top$$

where the subscript r denotes the first r columns/rows. The approximation error is:

$$\|\mathbf{A} - \mathbf{A}_r\|_F^2 = \sum_{i=r+1}^{\min(m,n)} \sigma_i^2, \quad \|\mathbf{A} - \mathbf{A}_r\|_2 = \sigma_{r+1}$$

This gives a principled measure of how much information is lost by truncating to rank r .

Relevance to neural networks.

- The condition number $\kappa = \sigma_1 / \sigma_{\min}$ measures how ill-conditioned the weight matrix is. Large $\kappa \Rightarrow$ training instability.

- Spectral normalisation (used in GANs, Volume II) normalises $\mathbf{W}$ so $\sigma_1(\mathbf{W}) = 1$, stabilising training by constraining the Lipschitz constant.
- **LoRA** (Hu et al., 2022): instead of fine-tuning a full $d \times d$ weight matrix, parameterise the update as $\Delta\mathbf{W} = \mathbf{B}\mathbf{A}$ with $\mathbf{B} \in \mathbb{R}^{d \times r}$, $\mathbf{A} \in \mathbb{R}^{r \times d}$, $\text{rank } r \ll d$. This is motivated by the observation that pre-trained weight matrices have low intrinsic rank when adapted to downstream tasks: the fine-tuning update $\Delta\mathbf{W}$ lives in a low-dimensional subspace of $\mathbb{R}^{d \times d}$.

EXAMPLE — SVD of a 2×2 weight matrix

Let $\mathbf{W} = \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}$. Since $\mathbf{W}$ is symmetric, the SVD equals the eigendecomposition. Eigenvalues: $\lambda_1 = 4$, $\lambda_2 = 2$. Eigenvectors: $\mathbf{v}_1 = (1, 1)^\top / \sqrt{2}$, $\mathbf{v}_2 = (1, -1)^\top / \sqrt{2}$. So $\sigma_1 = 4$, $\sigma_2 = 2$, condition number $\kappa = 2$. Rank-1 approximation: $\mathbf{W}_1 = 4 \cdot \mathbf{v}_1 \mathbf{v}_1^\top = 4 \cdot \frac{1}{2} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$. Frobenius error: $\|\mathbf{W} - \mathbf{W}_1\|_F = \sigma_2 = 2$.

0.11 Positive (Semi-)Definite Matrices

A symmetric matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is **positive definite** (PD) if $\mathbf{x}^\top \mathbf{A} \mathbf{x} > 0$ for all $\mathbf{x} \neq \mathbf{0}$, written $\mathbf{A} \succ 0$. It is **positive semi-definite** (PSD) if $\mathbf{x}^\top \mathbf{A} \mathbf{x} \geq 0$, written $\mathbf{A} \succeq 0$.

Equivalent Characterisations of PD Matrices

For a symmetric $\mathbf{A} \in \mathbb{R}^{n \times n}$, the following are equivalent:

- $\mathbf{A} \succ 0$ (all eigenvalues strictly positive).
- All principal minors of $\mathbf{A}$ are positive (Sylvester's criterion).
- $\mathbf{A} = \mathbf{L}\mathbf{L}^\top$ for some invertible lower-triangular $\mathbf{L}$ (Cholesky decomposition).
- $\mathbf{A} = \mathbf{B}^\top \mathbf{B}$ for some full-rank $\mathbf{B}$.

Role in neural networks.

- The Hessian $\mathbf{H} = \nabla^2 \mathcal{L}$ at a local minimum is PSD. It is PD when the minimum is non-degenerate (strict local minimum).
- The Fisher information matrix $\mathbf{F} = \mathbb{E}[\nabla \log p (\nabla \log p)^\top]$ is always PSD and governs natural gradient descent.
- The Gram matrix $\mathbf{G} = \mathbf{X}\mathbf{X}^\top$ (used in kernel methods and attention) is PSD by construction.
- Layer weight matrices need not be PSD; however, in normalised attention, $\mathbf{Q}\mathbf{K}^\top / \sqrt{d_k}$ can have indefinite spectrum — a source of training instability addressed by QK-normalisation (e.g. in Gemma 2).

EXAMPLE — Hessian at a Quadratic Minimum

Let $\mathcal{L}(\theta_1, \theta_2) = 3\theta_1^2 + 2\theta_1\theta_2 + 2\theta_2^2$. The Hessian:

$$\mathbf{H} = \begin{pmatrix} 6 & 2 \\ 2 & 4 \end{pmatrix}$$

Check PD: eigenvalues $\lambda_{1,2} = 5 \pm \sqrt{5}$, both positive. Alternatively, Sylvester: $\det(6) = 6 > 0$, $\det \mathbf{H} = 24 - 4 = 20 > 0$. Gradient descent on this quadratic converges in at most $n = 2$ steps with the exact line search (conjugate gradient property).

0.12 Information Theory Foundations

Three quantities from information theory are central to neural network loss functions and analysis.

Entropy, KL Divergence, Cross-Entropy

For discrete distributions $P = (p_1, \dots, p_K)$ and $Q = (q_1, \dots, q_K)$:

$$H(P) = - \sum_k p_k \log p_k \quad (\text{Shannon entropy, bits or nats})$$

$$D_{\text{KL}}(P\|Q) = \sum_k p_k \log \frac{p_k}{q_k} \quad (\text{KL divergence, always } \geq 0)$$

$$H(P, Q) = - \sum_k p_k \log q_k = H(P) + D_{\text{KL}}(P\|Q)$$

Derivation: Cross-Entropy as NLL. Given data with empirical distribution $\hat{P}$ (one-hot labels y) and model output distribution Q (softmax $\hat{y}$):

$$H(\hat{P}, Q) = - \sum_k y_k \log \hat{y}_k = - \log \hat{y}_c \quad (\text{categorical cross-entropy})$$

Minimising cross-entropy is equivalent to maximising the log-likelihood of Q under $\hat{P}$ — and to minimising $D_{\text{KL}}(\hat{P}\|Q)$ since $H(\hat{P})$ is constant with respect to model parameters.

Jensen's inequality. For a convex function ϕ and random variable X :

$$\phi(\mathbb{E}[X]) \leq \mathbb{E}[\phi(X)]$$

Setting $\phi = -\log$ (convex) yields $D_{\text{KL}}(P\|Q) \geq 0$ (Gibbs' inequality). This non-negativity is fundamental: it means cross-entropy always upper-bounds entropy, so minimising cross-entropy drives the model towards the data distribution.

EXAMPLE — Entropy and Cross-Entropy for a 3-Class Problem

True distribution: $P = (0.7, 0.2, 0.1)$. Model outputs: $Q = (0.6, 0.3, 0.1)$.

$$H(P) = -(0.7 \ln 0.7 + 0.2 \ln 0.2 + 0.1 \ln 0.1) = -(-0.357 - 0.322 - 0.230) = 0.909 \text{ nats}$$

$$H(P, Q) = -(0.7 \ln 0.6 + 0.2 \ln 0.3 + 0.1 \ln 0.1) = -(-0.356 - 0.241 - 0.230) = 0.827 \text{ nats}$$

Wait — cross-entropy must be $\geq$ entropy. Recalculate: $H(P) = 0.802$, $H(P, Q) = 0.827$. So $D_{\text{KL}} = 0.025$ nats: the model is slightly suboptimal.

Mutual information. The mutual information $I(X; Y) = D_{\text{KL}}(P(X, Y)\|P(X)P(Y))$ measures the dependence between random variables X and Y . In information-theoretic analyses of deep learning (Tishby & Schwartz-Ziv, 2017), mutual information $I(X; \mathbf{h}^{(l)})$ measures how much information the l -th layer's representation retains about the input, while $I(\mathbf{h}^{(l)}; Y)$ measures its relevance to the target — formalising the compression-prediction trade-off.

0.13 Convexity and the Optimization Landscape

Convex sets and functions. A set $\mathcal{C} \subseteq \mathbb{R}^n$ is convex if for all $\mathbf{x}, \mathbf{y} \in \mathcal{C}$ and $\lambda \in [0, 1]$: $\lambda\mathbf{x} + (1 - \lambda)\mathbf{y} \in \mathcal{C}$. A function $f: \mathcal{C} \rightarrow \mathbb{R}$ is convex if:

$$f(\lambda\mathbf{x} + (1 - \lambda)\mathbf{y}) \leq \lambda f(\mathbf{x}) + (1 - \lambda)f(\mathbf{y}), \quad \forall \mathbf{x}, \mathbf{y} \in \mathcal{C}, \lambda \in [0, 1]$$

Equivalently: f lies below any chord connecting two of its points. For differentiable f : f is convex $\iff f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{y} - \mathbf{x})$ for all $\mathbf{x}, \mathbf{y}$ (the tangent plane is a global lower bound). For twice-differentiable f : convex $\iff \mathbf{H}(\mathbf{x}) \succeq 0$ everywhere.

Why convexity matters. For convex f , every local minimum is a global minimum. Gradient descent with appropriate step size converges to the global minimum. For L -smooth convex functions (Lipschitz gradient with constant L):

$$f(\square_T) - f^* \leq \frac{L \|\square_0 - \square^*\|^2}{2T}$$

This $O(1/T)$ convergence guarantee is tight. For μ -strongly convex functions:

$$\|\square_T - \square^*\|^2 \leq \left(1 - \frac{\mu}{L}\right)^T \|\square_0 - \square^*\|^2$$

Linear convergence — exponentially fast. The ratio $\kappa = L/\mu$ (condition number) governs the rate: poorly conditioned problems ($\kappa \gg 1$) converge slowly.

Neural network loss landscapes are non-convex. A two-layer network $f(\mathbf{x}; \theta) = \mathbf{W}^{(2)}\phi(\mathbf{W}^{(1)}\mathbf{x})$ has a loss that is non-convex in θ : the permutation symmetry (swapping hidden neurons) creates multiple equivalent global minima. More importantly, the composition of non-linear functions with weight matrices makes the Hessian indefinite at most points.

Landscape property	Gradient descent guarantee	Example
Convex	Global minimum	Logistic regression, linear SVM
Strongly convex	Linear convergence rate	Ridge regression
Non-convex, no bad local min.	Global convergence (empirically)	Overparameterised NNs
Non-convex, with bad local min.	No guarantee	Sparse networks

Saddle points in high dimensions. A critical point $\square^*$ (where $\nabla f = \mathbf{0}$) is a saddle point if the Hessian has both positive and negative eigenvalues. In d dimensions, the probability that a random critical point is a local minimum (all Hessian eigenvalues positive) decreases exponentially with d under random matrix models (Dauphin et al., 2014). For $d = 10^8$ (large network), essentially all critical points encountered during training are saddle points, not local minima. Stochastic gradient noise allows escape from saddle points, making SGD preferable to full-batch gradient descent for non-convex optimisation.

The practical lesson: for sufficiently over-parameterised neural networks, gradient descent reliably finds good solutions not because the problem is convex, but because (a) saddle points are exponentially rare local minima, (b) most local minima have similar loss values, and (c) wider networks have fewer bad critical points (Du et al., 2019).

PART 1

The Single Neuron

We build the fundamental computational unit of every neural network from scratch: the artificial neuron. We derive its mathematical model, define every activation function, and compute their derivatives step by step.

1.1 The Mathematical Model

A *perceptron* is the elementary building block of every neural network: it takes an input vector, computes a weighted linear combination of it, adds a bias, and applies a nonlinear activation function.

Formal Definition of a Neuron

Given an input $\mathbf{x} \in \mathbb{R}^n$, a weight vector $\mathbf{w} \in \mathbb{R}^n$, a bias $b \in \mathbb{R}$, and an activation function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$, the neuron output is:

$$a = \sigma(z), \quad z = \mathbf{w}^\top \mathbf{x} + b = \sum_{i=1}^n w_i x_i + b$$

z is called the *pre-activation* or *logit*; a is the *activation*.

Why nonlinearity is indispensable. Without σ , a composition of L linear layers remains linear:

$$\mathbf{W}_L(\mathbf{W}_{L-1}(\cdots \mathbf{W}_1 \mathbf{x} \cdots)) = (\mathbf{W}_L \mathbf{W}_{L-1} \cdots \mathbf{W}_1) \mathbf{x} = \mathbf{W}_{\text{eff}} \mathbf{x}$$

Regardless of depth, the learned function is always a linear map, unable to separate non-linearly separable classes (e.g., XOR) or to approximate nonlinear functions.

Representational capacity. The nonlinearity creates *folds* in the feature space which, composed over many layers, can separate arbitrarily complex distributions. Intuitively: each layer transforms the input space so that the next layer finds the data more easily separable.

EXAMPLE — Complete Numerical Example

Neuron with $n = 3$: $\mathbf{w} = (0.5, -1.0, 0.3)^\top$, $b = 0.2$, $\sigma = \text{ReLU}$.

Input $\mathbf{x} = (1.5, 0.8, -0.5)^\top$:

$$z = 0.5 \cdot 1.5 + (-1.0) \cdot 0.8 + 0.3 \cdot (-0.5) + 0.2 = 0.75 - 0.80 - 0.15 + 0.20 = 0.00$$

$$a = \text{ReLU}(0.00) = \max(0, 0.00) = 0.00$$

Input $\mathbf{x}' = (2.0, 0.5, 1.0)^\top$:

$$z' = 0.5 \cdot 2.0 + (-1.0) \cdot 0.5 + 0.3 \cdot 1.0 + 0.2 = 1.0 - 0.5 + 0.3 + 0.2 = 1.0$$

$$a' = \text{ReLU}(1.0) = 1.0$$

Probabilistic interpretation. If σ is the sigmoid $\sigma(z) = 1/(1 + e^{-z})$, the output $a \in (0, 1)$ can be interpreted as $P(y = 1 \mid \mathbf{x}; \mathbf{w}, b)$, the posterior probability of the positive class given the input.

Connection to the biological neuron. The mathematical model is inspired by the McCulloch-Pitts neuron (1943): the soma integrates electrical signals from synapses (weights), and fires a spike (activation) if the sum exceeds a threshold (negative bias). The analogy is qualitative; artificial networks optimised with gradients are computationally distinct phenomena from neuroscience.

Parameters and Hyperparameters

- **Parameters** (learned during training): $\mathbf{w} \in \mathbb{R}^n, b \in \mathbb{R}$. A total of $n + 1$ values per neuron.
- **Hyperparameters** (chosen before training): activation type σ , architecture, learning rate.

For a layer with n_{in} inputs and n_{out} neurons: $n_{\text{in}} \cdot n_{\text{out}}$ weights + n_{out} biases = $n_{\text{out}}(n_{\text{in}} + 1)$ parameters.

Vector notation for a layer. For n_{out} neurons in parallel (a full layer):

$$\mathbf{a} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}), \quad \mathbf{W} \in \mathbb{R}^{n_{\text{out}} \times n_{\text{in}}}, \mathbf{b} \in \mathbb{R}^{n_{\text{out}}}$$

where σ is applied *element-wise*. This is the form that appears in code: `F.linear(x, W, b)` followed by `activation(z)`.

1.2 Activation Functions

An activation function $\phi : \mathbb{R} \rightarrow \mathbb{R}$ is applied element-wise after each linear transformation. Without nonlinearity, a composition of linear layers collapses to a single linear map: $\mathbf{W}^{(L)} \dots \mathbf{W}^{(1)}\mathbf{x} = \tilde{\mathbf{W}}\mathbf{x}$. Nonlinear activations are therefore the source of all representational power.

Desirable properties. An ideal activation should be:

- *Non-linear*: to break linearity.
- *Differentiable* (or at least sub-differentiable): for gradient-based training.
- *Non-saturating*: $|\phi'(x)| \not\rightarrow 0$ as $|x| \rightarrow \infty$, to avoid vanishing gradients.
- *Computationally efficient*: evaluated billions of times per training step.
- *Zero-centred output* (ideally): helps normalise activations through layers.

No activation satisfies all these simultaneously. The choice is a design trade-off.

The choice of activation function is among the most impactful architectural decisions: it determines gradient flow, convergence speed, and the expressive capacity of the network.

Main Activation Functions

- **Sigmoid**: $\sigma(z) = \frac{1}{1 + e^{-z}} \in (0, 1)$
- **Tanh**: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \in (-1, 1)$
- **ReLU**: $\text{ReLU}(z) = \max(0, z)$
- **Leaky ReLU**: $\text{LReLU}(z) = \max(\alpha z, z), \alpha = 0.01$

- **ELU:** $\text{ELU}(z) = \begin{cases} z & z \geq 0 \\ \alpha(e^z - 1) & z < 0 \end{cases}$
- **GELU:** $\text{GELU}(z) \approx z \cdot \Phi(z)$ where Φ is the standard normal CDF

Sigmoid. $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. The maximum derivative value is 0.25 (at $z = 0$), which causes *vanishing gradients* in deep networks: multiplying L sigmoid derivatives gives a factor $\leq 0.25^L$ that tends to zero exponentially. For networks with > 4 layers, sigmoid in hidden layers is discouraged; it remains ideal for binary outputs.

Tanh. Similar to sigmoid but zero-centred, which speeds up training (gradient updates have no systematic bias). $\tanh'(z) = 1 - \tanh^2(z) \leq 1$, with maximum 1 at $z = 0$. Still suffers from vanishing gradients for large inputs $|z| \gg 0$.

ReLU. Introduced by Nair & Hinton (2010) as a solution to vanishing gradients. $\text{ReLU}'(z) = \mathbb{1}[z > 0]$: the gradient is 1 for active units and 0 for inactive ones. The advantage is *sparse activation*: on average half the neurons are inactive, reducing feature interference. The risk is the *dying ReLU*: neurons that consistently receive negative inputs have zero gradient and never update.

EXAMPLE — Numerical Comparison of Activations

For $z \in \{-2, -1, 0, 1, 2\}$:

Activation	$z = -2$	$z = -1$	$z = 0$	$z = 1$	$z = 2$
Sigmoid	0.119	0.269	0.500	0.731	0.881
Tanh	-0.964	-0.762	0.000	0.762	0.964
ReLU	0	0	0	1	2
GELU	-0.045	-0.159	0.000	0.841	1.955

ELU. The ELU (Clevert et al., 2016) addresses the dying ReLU: for $z < 0$ it returns $\alpha(e^z - 1) < 0$, keeping the gradient non-zero ($\alpha e^z > 0$). The mean output tends to zero, which reduces bias shift in subsequent layers.

GELU. The GELU (Gaussian Error Linear Unit, Hendrycks & Gimpel, 2016) is used in BERT, GPT, and most modern Transformers. By combining the input with its normal CDF, it introduces a soft gate: small inputs are attenuated proportionally to their position in the distribution. It has no dead-neuron problem and approximates ReLU for large inputs.

Which activation to choose?

- **CNN hidden layers:** ReLU (simplicity, efficiency).
- **Transformers:** GELU or SiLU.
- **Binary classification output:** Sigmoid.
- **Multi-class classification output:** Softmax.
- **Regression output:** None (identity), or ReLU if the output must be non-negative.

Saturation and curvature. Sigmoid and tanh activations *saturate* for $|z| \gg 0$: the derivative approaches zero and the gradient vanishes. ReLU does not saturate for $z > 0$ (constant derivative = 1), which is the main reason for its success in deep networks.

1.3 Derivatives of Activation Functions — Step by Step

The backward pass requires evaluating $\phi'(z)$ at every pre-activation value stored during the forward pass. We derive each derivative from first principles.

Notation. We write $\sigma(z) = 1/(1 + e^{-z})$ for the logistic sigmoid. A key identity:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Proof. Let $u = 1 + e^{-z}$. Then $\sigma = 1/u$ and $du/dz = -e^{-z}$.

$$\sigma' = \frac{d}{dz} \frac{1}{u} = -\frac{1}{u^2} \cdot (-e^{-z}) = \frac{e^{-z}}{(1 + e^{-z})^2} = \sigma \cdot \frac{e^{-z}}{1 + e^{-z}} = \sigma(1 - \sigma)$$

Tanh derivative.

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1$$

$$\tanh'(z) = 1 - \tanh^2(z) = \text{sech}^2(z)$$

Proof. Let $s = e^z + e^{-z}$ (denominator) and $c = e^z - e^{-z}$ (numerator). Then $s' = e^z - e^{-z} = c$ and $c' = e^z + e^{-z} = s$. By quotient rule:

$$\tanh'(z) = \frac{s \cdot s - c \cdot c}{s^2} = \frac{s^2 - c^2}{s^2} = 1 - \left(\frac{c}{s}\right)^2 = 1 - \tanh^2(z)$$

Summary of activation derivatives:

Activation $\phi(z)$	Derivative $\phi'(z)$	Range of ϕ'
Sigmoid $\sigma(z)$	$\sigma(z)(1 - \sigma(z))$	$(0, 0.25]$
Tanh	$1 - \tanh^2(z)$	$(0, 1]$
ReLU $\max(0, z)$	$\mathbf{1}[z > 0]$	$\{0, 1\}$
Leaky ReLU	$\mathbf{1}[z > 0] + \alpha \mathbf{1}[z \leq 0]$	$\{\alpha, 1\}$
ELU	$\mathbf{1}[z > 0] + \alpha e^z \mathbf{1}[z \leq 0]$	$(0, 1]$
Softplus $\ln(1 + e^z)$	$\sigma(z)$	$(0, 1)$
GELU (approx)	$\Phi(z) + z\phi(z)$	$\approx (-0.09, 1.08]$

Vanishing gradient for sigmoid. The maximum of $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ is $1/4$ at $z = 0$ and decays to 0 for $|z| \gg 1$. For an L -layer network with sigmoid activations and pre-activations $|z| \gg 1$:

$$\left\| \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} \right\| \approx \left(\frac{1}{4} \right)^L \left\| \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(L)}} \right\| \rightarrow 0$$

For $L = 10$: $(0.25)^{10} \approx 10^{-6}$. Gradients for early layers are six orders of magnitude smaller than output gradients — making training the first layers essentially impossible.

EXAMPLE — Gradient Magnitude at Each Layer, Sigmoid vs. ReLU

4-layer network, output gradient = 1.0, assuming $\sigma'(z) \approx 0.2$ (typical mid-range), ReLU active fraction = 0.7.

Layer l	Sigmoid $ \delta^{(l)} $	ReLU $ \delta^{(l)} $	Ratio (ReLU/Sig)
4	1.000	1.000	1.0×
3	0.200	0.700	3.5×
2	0.040	0.490	12.3×
1	0.008	0.343	42.9×

At layer 1, ReLU gradients are $43\times$ larger, enabling effective learning in early layers.

Computing the derivatives of activation functions is the heart of backpropagation. Any error in the computation propagates through all preceding layers.

Sigmoid Derivative — Complete Derivation.

$$\sigma(z) = \frac{1}{1 + e^{-z}} = (1 + e^{-z})^{-1}$$

Applying the chain rule with $u = 1 + e^{-z}$:

$$\frac{d\sigma}{dz} = -u^{-2} \cdot \frac{du}{dz} = -(1 + e^{-z})^{-2} \cdot (-e^{-z}) = \frac{e^{-z}}{(1 + e^{-z})^2}$$

Rewriting in terms of $\sigma(z)$:

$$\frac{e^{-z}}{(1 + e^{-z})^2} = \frac{1}{1 + e^{-z}} \cdot \frac{e^{-z}}{1 + e^{-z}} = \sigma(z) \cdot (1 - \sigma(z))$$

$$\boxed{\sigma'(z) = \sigma(z)(1 - \sigma(z))}$$

This form is computationally efficient: if $a = \sigma(z)$ has already been computed in the forward pass, the derivative is simply $a(1 - a)$.

Tanh Derivative.

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Using the quotient rule:

$$\tanh'(z) = \frac{(e^z + e^{-z})^2 - (e^z - e^{-z})^2}{(e^z + e^{-z})^2} = 1 - \tanh^2(z)$$

$$\boxed{\tanh'(z) = 1 - \tanh^2(z) = \operatorname{sech}^2(z)}$$

ReLU Derivative.

$$\operatorname{ReLU}(z) = \max(0, z) = \begin{cases} z & z > 0 \\ 0 & z < 0 \end{cases}$$

$$\operatorname{ReLU}'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases} = \mathbb{1}[z > 0]$$

At $z = 0$, ReLU is not differentiable. In practice a *subgradient* is used: conventionally 0 (PyTorch) or 0.5 (some implementations).

Leaky ReLU Derivative.

$$\operatorname{LReLU}'(z) = \begin{cases} 1 & z > 0 \\ \alpha & z < 0 \end{cases}$$

With $\alpha = 0.01$: the gradient never dies, but for $z < 0$ it is scaled by α .

EXAMPLE — Derivative Computation in Backpropagation

Sigmoid layer in the forward pass: $z = 1.5$, $a = \sigma(1.5) = 0.818$.

In the backward pass, suppose $\delta = \partial\mathcal{L}/\partial a = -0.4$ (upstream gradient). Then:

$$\frac{\partial\mathcal{L}}{\partial z} = \delta \cdot \sigma'(z) = \delta \cdot a(1-a) = -0.4 \cdot 0.818 \cdot 0.182 = -0.0596$$

The gradient is *attenuated* by a factor $0.818 \times 0.182 \approx 0.149$. With $L = 20$ sigmoid layers, the cumulative factor would be $\approx 0.149^{20} \approx 10^{-18}$: the gradient has effectively vanished.

Vector Derivatives (Element-Wise)

For a layer with σ applied element-wise, the Jacobian is diagonal:

$$\frac{\partial \mathbf{a}}{\partial \mathbf{z}} = \text{diag}(\sigma'(z_1), \sigma'(z_2), \dots, \sigma'(z_n)) \in \mathbb{R}^{n \times n}$$

Since it is diagonal, the product $(\partial \mathbf{a} / \partial \mathbf{z})^\top \delta \mathbf{0}$ is simply the element-wise multiplication $\sigma'(\mathbf{z}) \odot \delta \mathbf{0}$, reducing complexity from $O(n^2)$ to $O(n)$.

1.4 The GELU Activation Function

GELU (Gaussian Error Linear Unit) is the activation function used in BERT, GPT, and most modern Transformers:

$$\text{GELU}(z) \triangleq z \cdot \Phi(z), \quad \Phi(z) = \frac{1}{2} \left[1 + \text{erf}\left(\frac{z}{\sqrt{2}}\right) \right]$$

Intuition. $\Phi(z)$ is the probability that $N \sim \mathcal{N}(0, 1)$ satisfies $N \leq z$. So GELU weights the input by how likely it is to be “activated” under a Gaussian gate.

Derivative:

$$\text{GELU}'(z) = \Phi(z) + z \cdot \phi(z), \quad \phi(z) = \frac{1}{\sqrt{2\pi}} e^{-z^2/2}$$

Practical approximation:

$$\text{GELU}(z) \approx 0.5z \left[1 + \tanh\left(\sqrt{2/\pi} (z + 0.044715 z^3)\right) \right]$$

EXAMPLE — GELU numerical values

z	$\Phi(z)$	$\text{GELU}(z)$	$\text{GELU}'(z)$
−2.0	0.0228	−0.0455	0.0857
−1.0	0.1587	−0.1587	0.3293
0.0	0.5000	0.0000	0.5000
1.0	0.8413	0.8413	1.0840
2.0	0.9772	1.9544	1.0850

1.5 Swish and SiLU

$$\text{Swish}(z) \triangleq z \cdot \sigma(z) = \frac{z}{1 + e^{-z}}$$

Proposed by Ramachandran et al. (2017) via automated search over activation functions; re-discovered as the *Sigmoid Linear Unit* (SiLU) by Elfwang et al. (2018). The two are identical.

Derivative. Using the product rule $\text{Swish}'(z) = \sigma(z) + z\sigma'(z) = \sigma(z) + z\sigma(z)(1 - \sigma(z))$:

$$\text{Swish}'(z) = \sigma(z)(1 + z(1 - \sigma(z)))$$

Let $s = \sigma(z)$: $\text{Swish}'(z) = s + zs(1 - s) = s(1 + z - zs)$.

Key properties.

- *Non-monotone*: for $z \approx -1.28$, $\text{Swish}(z) \approx -0.278$ (local minimum). The slight negative region allows gradient flow for small negative pre-activations.
- *Smooth everywhere*: unlike ReLU, no kink at $z = 0$. Second derivatives exist, enabling second-order optimisation.
- *Self-gating*: $\text{Swish}(z) = z \cdot \sigma(z)$. The sigmoid factor $\sigma(z)$ acts as a soft gate: for $z \gg 0$, $\sigma(z) \rightarrow 1$ and $\text{Swish}(z) \rightarrow z$ (linear); for $z \ll 0$, $\sigma(z) \rightarrow 0$ and $\text{Swish}(z) \rightarrow 0$ (suppressed).
- *Unbounded above, bounded below*: like ReLU but avoids dying neuron problem at $z \ll 0$ due to $\sigma(z) \rightarrow 0$ slowly.

EXAMPLE — Swish vs. ReLU at $z = -0.5, 0, 0.5, 2$

z	$\sigma(z)$	$\text{Swish}(z)$	$\text{ReLU}(z)$	$\text{Swish}'(z)$
-0.5	0.378	-0.189	0.000	0.072
0.0	0.500	0.000	0.000	0.500
0.5	0.622	0.311	0.500	0.928
2.0	0.880	1.762	2.000	1.092

Note: $\text{Swish}'(2.0) = 1.092 > 1$, meaning the activation slightly amplifies large positive values — a property ReLU lacks with its constant gradient of 1.

Swish is the default activation in EfficientNet and is used in many transformer variants (PaLM uses SwiGLU = $\text{Swish}(xW) \odot (xV)$, a gated variant).

1.6 Universal Approximation Theorem

Theorem (Cybenko 1989, Hornik 1991)

For any continuous $f : [0, 1]^n \rightarrow \mathbb{R}$ and $\epsilon > 0$, there exist N and parameters $\{w_i, b_i, c_i\}$ such that:

$$\sup_{\mathbf{x} \in [0, 1]^n} \left| f(\mathbf{x}) - \sum_{i=1}^N c_i \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i) \right| < \epsilon$$

Proof sketch (width version). The argument proceeds via three steps. First, a single sigmoid neuron $\sigma(\mathbf{w}^\top \mathbf{x} + b)$ with large $|w|$ approximates an indicator function $\mathbf{1}[\mathbf{w}^\top \mathbf{x} \geq -b]$: a “half-space” classifier. Second, sums of two such neurons with opposite signs and slightly shifted biases approximate a “bump” function localised to a thin slab:

$$B_\delta(\mathbf{x}) \approx \sigma(\mathbf{w}\mathbf{n}^\top \mathbf{x} + b_1) - \sigma(\mathbf{w}\mathbf{n}^\top \mathbf{x} + b_2), \quad b_2 - b_1 = w\delta.$$

Third, a sufficiently fine grid of such bumps over $[0, 1]^n$ approximates any continuous f via its piecewise-constant interpolant. Since f is uniformly continuous on the compact domain, the approximation error vanishes as the grid is refined, and N grows at most polynomially in $1/\epsilon$ for fixed n .

The theorem guarantees existence, not learnability. Finding the N neurons and their weights from data is a separate (and much harder) problem. The theorem says a shallow network can express f ; it says nothing about whether gradient descent can find the right parameters, or how many training examples are required.

Width vs. depth. The classical theorem concerns one hidden layer with unbounded width. The *depth* version (Lu et al., 2017) shows that width- $(n + 4)$ networks of sufficient depth also universally approximate continuous functions on $\mathbb{R}^n$, where n is the input dimension. Width is exchangeable with depth up to polynomial cost.

Depth separation. Eldan & Shamir (2016) exhibited a radial function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ that a 2-layer network requires $\Omega(\exp(n))$ neurons to approximate to constant error, while a 3-layer network needs only $O(n^2)$ neurons. More generally, for certain classes of functions, d -layer networks require exponentially fewer neurons than $(d-1)$ -layer networks — this is the *depth-efficiency* advantage and the theoretical basis for deep (rather than shallow) architectures.

EXAMPLE — Approximating $f(x) = \sin(2\pi x)$ on $[0, 1]$

A shallow network with $N = 20$ neurons ($\sigma = \tanh$) can approximate $\sin(2\pi x)$ to precision $\epsilon < 0.01$ by fitting 20 shifted sigmoids. A deeper network with 2 layers of 5 neurons each achieves comparable precision, because the first layer can represent low-frequency features and the second combines them. In practice, depth reduces the parameter count needed for a given approximation quality.

1.7 Activation Function Summary and Selection Guide

Having derived every activation function from first principles, we now compare them systematically. The choice of activation has significant practical consequences for training speed, representational capacity, and gradient flow.

Function	Formula	Range	$f'(x)$ range	Dead neurons?
Sigmoid	$1/(1 + e^{-x})$	$(0, 1)$	$(0, 0.25]$	No (but saturates)
Tanh	$(e^x - e^{-x})/(e^x + e^{-x})$	$(-1, 1)$	$(0, 1]$	No (but saturates)
ReLU	$\max(0, x)$	$[0, \infty)$	$\{0, 1\}$	Yes ($x < 0$)
Leaky ReLU	$\max(\alpha x, x)$	$\mathbb{R}$	$\{\alpha, 1\}$	No
GELU	$x\Phi(x)$	$\mathbb{R}$	smooth, > 0 mostly	Rare
Swish/SiLU	$x\sigma(x)$	$\approx (-0.28, \infty)$	smooth	Rare

Saturation and the vanishing gradient. Sigmoid and tanh saturate (output near ± 1 or $0/1$) for large $|x|$: their derivatives approach 0. In deep networks, products of near-zero derivatives cause vanishing gradients. ReLU avoids saturation for $x > 0$ (derivative exactly 1) but can “die” for $x < 0$.

Practical selection rules.

- **Hidden layers of MLPs/CNNs:** ReLU is the default. Fast, sparse, no saturation. Use

He initialisation.

- **Transformers (FFN layers):** GELU or SiLU. Smooth approximation to ReLU; empirically better for language modelling.
- **Gates in LSTM/GRU:** Sigmoid (output $\in (0, 1)$, interpretable as probability). Tanh for candidate state.
- **Output layer:** depends on task — sigmoid for binary classification, softmax for multi-class, linear for regression.
- **Avoid:** sigmoid/tanh in hidden layers of deep networks (saturation causes training instability).

Biological plausibility. The biological neuron fires (action potential) when its membrane potential exceeds a threshold — a step function. The sigmoid was originally proposed as a smooth approximation to this threshold. However, biological neurons are far more complex: they exhibit spike-rate coding, temporal dynamics, and dendritic computation. Modern artificial activations (ReLU, GELU) bear little biological resemblance but are computationally effective. The field of neuromorphic computing studies hardware implementations of spiking neural networks (SNNs) with more biologically faithful discrete spike signals, but these are not yet competitive with standard deep learning on most benchmarks.

Dying ReLU: diagnosis and cure.

- *Diagnosis:* monitor the fraction of zero activations per layer during training. If $> 50\%$ are consistently zero for a layer, neurons may be dead.
- *Causes:* large learning rates causing large negative weight updates; poor initialisation.
- *Cures:* Leaky ReLU ($\alpha = 0.01$); ELU; lower learning rate; proper He initialisation; gradient clipping.

Multi-Layer Perceptron: Architecture and Forward Pass

We stack neurons into layers, define the complete notation for an L -layer network, and trace an input vector through the network step by step.

2.1 Architecture and Complete Notation

A *multi-layer neural network* (MLP, Multi-Layer Perceptron) is a composition of affine transformations followed by nonlinearities. Establishing a rigorous and consistent notation is essential before proceeding to the forward and backward passes.

Notation for an L -Layer MLP

- L : number of layers (excluding the input).
- $n^{[l]}$: number of neurons in layer l , with $n^{[0]} = n_{\text{in}}$ (input) and $n^{[L]} = n_{\text{out}}$ (output).
- $\mathbf{W}^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$: weight matrix of layer l .
- $\mathbf{b}^{[l]} \in \mathbb{R}^{n^{[l]}}$: bias of layer l .
- $\mathbf{z}^{[l]} \in \mathbb{R}^{n^{[l]}}$: pre-activation of layer l .
- $\mathbf{a}^{[l]} \in \mathbb{R}^{n^{[l]}}$: activation of layer l , with $\mathbf{a}^{[0]} = \mathbf{x}$.
- $\sigma^{[l]}$: activation function of layer l .

Forward pass equations (single sample).

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]} \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}, \quad l = 1, \dots, L$$

$$\mathbf{a}^{[l]} = \sigma^{[l]}(\mathbf{z}^{[l]}), \quad l = 1, \dots, L$$

The network output is $\hat{\mathbf{y}} = \mathbf{a}^{[L]}$.

Parameter count. The total number of parameters in an MLP is:

$$P = \sum_{l=1}^L (n^{[l]} \cdot n^{[l-1]} + n^{[l]}) = \sum_{l=1}^L n^{[l]}(n^{[l-1]} + 1)$$

EXAMPLE — 784-256-128-10 Architecture

MLP for MNIST with $n^{[0]} = 784$, $n^{[1]} = 256$, $n^{[2]} = 128$, $n^{[3]} = 10$:

- Layer 1: $256 \times 784 + 256 = 200,960$ parameters
- Layer 2: $128 \times 256 + 128 = 32,896$ parameters
- Layer 3 (output): $10 \times 128 + 10 = 1,290$ parameters
- **Total:** 235,146 parameters

With float32, the memory for weights is $235,146 \times 4 \approx 940$ KB.

Batch notation. For a mini-batch of B samples, the activations become matrices:

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \mathbf{A}^{[l-1]} + \mathbf{b}^{[l]} \mathbf{1}^\top \in \mathbb{R}^{n^{[l]} \times B}$$

where the bias $\mathbf{b}^{[l]}$ is added to every column via broadcasting.

Index convention. Some texts use $\mathbf{h}^{(l)}$ for hidden activations, reserving $\mathbf{a}$ for the output. Throughout this volume we always use $\mathbf{a}^{[l]}$ for consistency with the modern literature (Goodfellow et al., 2016; He et al., 2016).

2.2 Forward Propagation

Forward propagation is the sequential application of the layer equations from $l = 1$ to $l = L$.

Algorithm: Forward Pass

Input: $\mathbf{x} \in \mathbb{R}^{n_0}$, parameters $\{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^L$

For $l = 1, 2, \dots, L$:

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$$

$$\mathbf{a}^{(l)} = \phi^{(l)}(\mathbf{z}^{(l)})$$

Output: $\hat{\mathbf{y}} = \mathbf{a}^{(L)}$

Computational cost. Each matrix-vector product $\mathbf{W}^{(l)} \mathbf{a}^{(l-1)}$ costs $O(n_l \cdot n_{l-1})$ multiplications. Total cost: $O(\sum_l n_l n_{l-1})$.

2.3 Complete Numerical Example — Forward Pass

Network: $n_0 = 3$, $n_1 = 2$, $n_2 = 1$, sigmoid activations throughout.

$$\mathbf{W}^{(1)} = \begin{pmatrix} 0.1 & 0.3 & -0.5 \\ 0.2 & -0.4 & 0.6 \end{pmatrix}, \quad \mathbf{b}^{(1)} = \begin{pmatrix} 0.0 \\ 0.1 \end{pmatrix}$$

$$\mathbf{W}^{(2)} = (0.7 \quad -0.8), \quad \mathbf{b}^{(2)} = (-0.2)$$

$$\mathbf{x} = \begin{pmatrix} 1.0 \\ 0.5 \\ -1.0 \end{pmatrix}$$

Layer 1 — pre-activation:

$$z_1^{(1)} = (0.1)(1.0) + (0.3)(0.5) + (-0.5)(-1.0) + 0.0 = 0.1 + 0.15 + 0.5 = 0.75$$

$$z_2^{(1)} = (0.2)(1.0) + (-0.4)(0.5) + (0.6)(-1.0) + 0.1 = 0.2 - 0.2 - 0.6 + 0.1 = -0.5$$

$$\mathbf{z}^{(1)} = \begin{pmatrix} 0.75 \\ -0.5 \end{pmatrix}$$

Layer 1 — activation:

$$a_1^{(1)} = \sigma(0.75) = \frac{1}{1 + e^{-0.75}} = \frac{1}{1 + 0.4724} \approx 0.6792$$

$$a_2^{(1)} = \sigma(-0.5) = \frac{1}{1 + e^{0.5}} = \frac{1}{1 + 1.6487} \approx 0.3775$$

$$\mathbf{a}^{(1)} = \begin{pmatrix} 0.6792 \\ 0.3775 \end{pmatrix}$$

Layer 2 — pre-activation:

$$z_1^{(2)} = (0.7)(0.6792) + (-0.8)(0.3775) + (-0.2) = 0.4754 - 0.3020 - 0.2 = -0.0266$$

Layer 2 — activation:

$$\hat{y} = a_1^{(2)} = \sigma(-0.0266) = \frac{1}{1 + e^{0.0266}} \approx \frac{1}{1 + 1.0269} \approx 0.4934$$

The network output is $\hat{y} \approx 0.4934$ (probability $\approx 49.3\%$ of class 1).

2.4 Vectorised Batch Forward Pass

Stack B input vectors as rows: $\mathbf{X} \in \mathbb{R}^{B \times n_0}$. The forward pass becomes:

$$\mathbf{Z}^{(l)} = \mathbf{A}^{(l-1)}\mathbf{W}^{(l)\top} + \mathbf{1}_B\mathbf{b}^{(l)\top} \in \mathbb{R}^{B \times n_l}, \quad \mathbf{A}^{(l)} = \phi^{(l)}(\mathbf{Z}^{(l)})$$

The dominant cost is the matrix-matrix product $O(B \cdot n_l \cdot n_{l-1})$, optimal for GPU Tensor Cores.

Broadcasting. The bias term $\mathbf{1}_B\mathbf{b}^{(l)\top}$ uses broadcasting: $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$ is added to every row of $\mathbf{Z}^{(l)}$. In NumPy/PyTorch this is $\mathbf{Z} = \mathbf{A}@\mathbf{W}.T + \mathbf{b}$, where $\mathbf{b}$ broadcasts over the batch dimension automatically.

Activation element-wise. $\phi^{(l)}$ applies independently to each entry: $A_{bi}^{(l)} = \phi^{(l)}(Z_{bi}^{(l)})$. For a batch, this is a single vectorised operation at native hardware speed.

Why batch processing?

- *Hardware utilisation:* a GPU has thousands of CUDA cores. A single forward pass ($B = 1$) leaves most idle. With $B = 256$, the matrix product $\mathbf{XW}^\top$ occupies all cores.
- *Gradient stability:* the gradient $\nabla_{\mathbf{W}}\mathcal{L} = \frac{1}{B}\mathbf{A}^{(l-1)\top}\bar{\mathbf{Z}}^{(l)}$ averages over B examples, reducing variance by $1/\sqrt{B}$ versus a single-example gradient.
- *BatchNorm dependency:* batch normalisation requires $B \geq 2$ (and works best with $B \geq 32$) to estimate mean and variance within each mini-batch.

EXAMPLE — Batch Forward Pass, $B = 4, n_0 = 3 \rightarrow n_1 = 2$

$$\mathbf{X} = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix}, \mathbf{W} = \begin{pmatrix} 0.5 & -0.5 & 1 \\ 1 & 0.5 & -1 \end{pmatrix}, \mathbf{b} = (0.1, -0.1)^\top.$$

$\mathbf{Z}^{(1)} = \mathbf{XW}^\top + \mathbf{1}_4\mathbf{b}^\top$. Row 1: $(0.5 \cdot 1 - 0.5 \cdot 0 + 1 \cdot 2, 1 \cdot 1 + 0.5 \cdot 0 - 1 \cdot 2) + (0.1, -0.1) = (2.6, -1.1)$.

4 matrix rows computed in parallel on GPU vs. 4 sequential calls on CPU. For a real LLM with $n = 4096$ and $B = 512$: $512 \times 4096 \times 4096 = 8.6 \times 10^9$ multiplications — completed in ~ 28 ms on an A100.

2.5 Depth, Width, and Representational Power

Property	Increasing width	Increasing depth
Approximation power	Polynomial	Exponential
Gradient stability	Better (shorter paths)	Worse (more products)
Parameter efficiency	Lower	Higher
Feature hierarchy	Not learned	Learned

Formal depth-width separation. Eldan & Shamir (2016) proved that there exist functions computable by a depth-3 network with $O(n)$ neurons that require exponentially many neurons ($2^{\Omega(n)}$) to compute with depth 2. This is an exponential depth-width trade-off: going from 3 layers to 2 layers requires exponentially more width.

Feature hierarchy in practice. In CNNs trained on ImageNet (Zeiler & Fergus, 2014):

- Layer 1: oriented edges and colour blobs (Gabor-like filters).
- Layer 2: textures and simple shapes (corners, curves).
- Layer 3: object parts (eyes, wheels, fur patterns).
- Layer 4–5: high-level semantic concepts (faces, text, specific object classes).

This hierarchy emerges from training, not design — depth enables compositional feature learning.

Parameter count comparison. To represent a function with n input variables, k breakpoints per dimension:

- Lookup table (no network): k^n entries — exponential in n .
- Width- W single hidden layer: Wn parameters.
- Depth- L network with width W : LW^2 parameters for $O(W^L)$ combinatorial regions in input space.

Deep networks compute exponentially many linear regions with only polynomially many parameters (Montufar et al., 2014).

EXAMPLE — Linear Regions of ReLU Networks

A ReLU network with n_0 inputs and L hidden layers of width n has at most:

$$\sum_{j=0}^{n_0} \binom{nL}{j}$$

linear regions (Zaslavsky's theorem applied to hyperplane arrangements). For $n_0 = 2$, $n = 4$, $L = 3$: upper bound $\approx \sum_{j=0}^2 \binom{12}{j} = 1 + 12 + 66 = 79$ regions. In practice, exponentially many of these are activated by a typical dataset, giving the network high expressivity.

2.6 Forward Mode vs. Reverse Mode Automatic Differentiation

Forward mode propagates derivatives of a scalar input from input to output. Cost: $O(n_{\text{params}})$ per output scalar.

Reverse mode (backpropagation) propagates derivatives of a scalar output backwards. Cost: $O(n_{\text{params}})$ per scalar loss — computes all parameter gradients in one pass at $\approx 3\times$ the forward pass cost.

Neural network training has 1 scalar output (loss) and millions of inputs (parameters). Reverse mode computes all gradients simultaneously; forward mode would require one pass per parameter — completely intractable.

Formal cost comparison. Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$.

- *Forward mode*: propagates Jacobian-vector products $J\mathbf{v}$; cost per direction: $O(C_f)$. Full Jacobian: n passes.
- *Reverse mode*: propagates vector-Jacobian products $\mathbf{u}^\top J$; cost per output: $\approx 3C_f$. Full Jacobian: m passes.

For deep learning $n \sim 10^8$, $m = 1$: reverse mode computes all gradients in $\approx 3C_f$ vs. $n \cdot C_f$ for forward mode. Speedup: $n/3 \approx 3 \times 10^7$. Forward mode is still useful for Hessian-vector products $H\mathbf{v}$, computed as a reverse pass through a forward-mode pass (used in K-FAC and conjugate gradient second-order methods).

2.7 Memory Complexity of Forward and Backward Passes

Training a network requires storing intermediate activations for use in the backward pass. This is the dominant memory cost for deep models.

Forward pass memory. For an L -layer network with widths $n_0, n_1, \dots, n_L$ and batch size B , storing all activations requires:

$$M_{\text{fwd}} = B \sum_{l=0}^L n_l \quad (\text{floats})$$

For a Transformer with $L = 12$ layers, hidden size $d = 768$, $T = 512$ tokens, $B = 32$:

$$M_{\text{fwd}} = 32 \times 512 \times 768 \times 12 \approx 150 \text{ M floats} = 600 \text{ MB (float32)}$$

Parameter memory. Separate from activations: $\sum_l (n_l n_{l-1} + n_l)$ parameters. For BERT-base: 110 M parameters = 440 MB (float32) or 220 MB (bfloat16).

Gradient checkpointing. Trade compute for memory by not storing all activations during the forward pass. Instead, recompute them during the backward pass from stored *checkpoints* at regular intervals. With checkpoints every k layers:

- Memory cost: $O(\sqrt{L})$ activations instead of $O(L)$.
- Compute cost: one extra forward pass ($\approx 33\%$ overhead).

Used in training very large models (GPT-3, LLaMA) where GPU VRAM is the bottleneck.

EXAMPLE — Memory for a 3-Layer MLP, $B = 64$

Architecture: $n_0 = 784$ (MNIST), $n_1 = 512$, $n_2 = 256$, $n_3 = 10$.

Activations: $64(784 + 512 + 256 + 10) = 64 \times 1562 = 99,968$ floats ≈ 400 KB.

Parameters: $784 \times 512 + 512 \times 256 + 256 \times 10 = 401,408 + 130,560 + 2,560 = 535,528$ floats ≈ 2.1 MB.

Gradients: same shape as parameters = 2.1 MB.

Adam states: $\mathbf{m}$ and $\mathbf{v}$ each = 2.1 MB.

Total: ≈ 9 MB — trivial for modern GPUs (16–80 GB VRAM), but scale to LLMs with $P \sim 10^{10}$ parameters and the same analysis gives ~ 400 GB just for parameters+states.

2.8 Computational Graph and Dynamic vs. Static Computation

A **computational graph** is a directed acyclic graph (DAG) where nodes represent operations or values, and edges represent data dependencies. Backpropagation is equivalent to traversing this graph in reverse topological order.

Static graphs (TensorFlow 1.x, JAX by default) are compiled before execution. Benefits: compiler optimisations (kernel fusion, operator reordering), ahead-of-time shape checking. Drawback: control flow (loops, conditionals) must be expressed as graph primitives.

Dynamic graphs (PyTorch, JAX with eager mode) build the graph on the fly during the forward pass. Benefits: Python control flow works naturally (variable-length sequences, conditional architectures). Drawback: less opportunity for static optimisation.

PyTorch's `torch.compile()` (since 2.0) provides dynamic graphs with JIT compilation, closing most of the performance gap with static frameworks. The underlying compiler (TorchInductor) uses Triton to generate optimised GPU kernels.

2.9 Parameter Sharing, Equivariance, and Invariance

A key architectural principle in deep learning is building *structure* into the network's parameterisation to exploit known properties of the data. This reduces parameters, improves generalisation, and can dramatically reduce data requirements.

Parameter sharing. In a standard MLP, every connection between two layers has its own weight. *Parameter sharing* constrains certain weights to be equal. The most important example is the convolutional layer: the same kernel $\mathbf{w} \in \mathbb{R}^{k \times k}$ is applied at every spatial position, reducing the parameter count from $H \times W \times C_{\text{in}} \times C_{\text{out}}$ (fully connected) to $k^2 \times C_{\text{in}} \times C_{\text{out}}$.

Equivariance. A function f is equivariant to a transformation T if:

$$f(T(\mathbf{x})) = T'(f(\mathbf{x}))$$

where T' is the corresponding transformation of the output. Convolutional layers are equivariant to translation: if the input is shifted by (δ_x, δ_y) , the output feature map shifts by the same amount. This means that a feature detector (e.g., an edge detector) learned at one position automatically works at all positions.

Invariance. A function f is invariant to T if $f(T(\mathbf{x})) = f(\mathbf{x})$ — the output is unchanged. Global average pooling makes a CNN invariant to spatial location: pooling over all positions produces the same channel statistics regardless of where a feature appears.

Architecture	Symmetry	Property	How achieved
CNN	Translation	Equivariant	Shared conv. kernel
CNN + GAP	Translation	Invariant	Global avg. pool
RNN	Time shift	Equivariant	Shared recurrent weights
Transformer	Permutation	Equivariant (without PE)	Self-attention
PointNet	Permutation	Invariant	Max pooling over points

Why equivariance helps. Suppose f is equivariant to a group G of transformations, and the true labelling function f^* is invariant to G . Then the combined map $f^* \circ f$ is also invariant — the invariance is built into the pipeline. This means the network only needs to learn the quotient of the input space under G , which may be much smaller, requiring less data and fewer parameters.

Group equivariant networks (G-CNNs). Cohen & Welling (2016) generalised CNNs to be equivariant to larger symmetry groups (rotations, reflections) by designing kernels that transform appropriately under the group action. For medical imaging (rotationally symmetric), G-CNNs outperform standard CNNs with far fewer parameters.

EXAMPLE — Counting CNN Parameters vs. Fully Connected

Input: 28×28 grayscale image. First layer: 32 filters of size 3×3 .

- Fully connected: $28 \times 28 \times 32 = 25,088$ parameters.
- CNN (shared kernel): $3 \times 3 \times 1 \times 32 + 32$ (biases) = 320 parameters.
- Reduction: $78\times$ fewer parameters, with guaranteed translation equivariance.

Loss Functions

A loss function quantifies the discrepancy between prediction and target. We derive MSE, binary cross-entropy, and categorical cross-entropy from probabilistic first principles.

3.1 Mean Squared Error (MSE)

The Mean Squared Error is the canonical regression loss. For predictions $\hat{y}_i$ and targets y_i , $i = 1, \dots, N$:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

Probabilistic derivation. MSE is the negative log-likelihood of a Gaussian observation model. If $y_i = f_\theta(\mathbf{x}_i) + \epsilon_i$ with $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ i.i.d., then:

$$-\log \mathcal{L}(\theta) = \frac{N}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

Maximising $\mathcal{L}$ (MLE) is equivalent to minimising $\mathcal{L}_{\text{MSE}}$ — the Gaussian noise model justifies MSE for regression.

Bias-variance decomposition. For a fixed test input $\mathbf{x}^*$, the expected MSE over random training datasets $\mathcal{D}$:

$$\mathbb{E}_{\mathcal{D}}[(y^* - \hat{y})^2] = \underbrace{(\mathbb{E}_{\mathcal{D}}[\hat{y}] - f(\mathbf{x}^*))^2}_{\text{Bias}^2} + \underbrace{\mathbb{E}_{\mathcal{D}}[(\hat{y} - \mathbb{E}_{\mathcal{D}}[\hat{y}])^2]}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Irreducible noise}} \quad \left| \begin{array}{l} \{Z\} \\ \{Z\} \end{array} \right.$$

- **Bias:** error from wrong assumptions (underfitting). High for underparameterised models.
- **Variance:** sensitivity to training data (overfitting). High for overparameterised models without regularisation.
- **Irreducible noise:** inherent randomness in y^* given $\mathbf{x}^*$.

Modern deep networks are vastly overparameterised yet generalise well (“double descent” phenomenon): variance re-decreases past the interpolation threshold as the network finds the minimum-norm interpolant.

The *Mean Squared Error* is the standard loss function for regression problems. It is derived from maximum likelihood under a Gaussian noise model.

MSE Definition

For a batch of N samples with targets $y_i \in \mathbb{R}$ and predictions $\hat{y}_i \in \mathbb{R}$:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

In vector form: $\mathcal{L}_{\text{MSE}} = \frac{1}{N} \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$.

Derivation from Gaussian likelihood. Suppose each target is generated as $y_i = f(\mathbf{x}_i; \theta) + \epsilon_i$ with $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$. The log-likelihood is:

$$\log p(\mathcal{D}; \theta) = -\frac{N}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

Maximising the log-likelihood is equivalent to minimising $\sum_i (y_i - \hat{y}_i)^2$, i.e., the MSE.

Gradient of MSE with respect to the output.

$$\frac{\partial \mathcal{L}_{\text{MSE}}}{\partial \hat{y}_i} = -\frac{2}{N} (y_i - \hat{y}_i) = \frac{2}{N} (\hat{y}_i - y_i)$$

The gradient is proportional to the *residual* $\hat{y}_i - y_i$: if the prediction is too high, the gradient pushes it downward.

EXAMPLE — Numerical Computation

Predictions $\hat{\mathbf{y}} = (2.5, 0.0, 2.0, 8.0)^\top$, targets $\mathbf{y} = (3.0, -0.5, 2.0, 7.0)^\top$:

$$\begin{aligned} \mathcal{L}_{\text{MSE}} &= \frac{1}{4} [(3.0 - 2.5)^2 + (-0.5 - 0.0)^2 + (2.0 - 2.0)^2 + (7.0 - 8.0)^2] \\ &= \frac{1}{4} [0.25 + 0.25 + 0.00 + 1.00] = \frac{1.50}{4} = 0.375 \end{aligned}$$

Sensitivity to outliers. MSE penalises large errors quadratically: a residual of 10 contributes 100 to the loss, vs 1 for a residual of 1. This makes it sensitive to outliers. Alternatives: MAE (L^1), Huber loss (Section 3.4).

RMSE and scale. Root MSE $\text{RMSE} = \sqrt{\mathcal{L}_{\text{MSE}}}$ has the same units as the target, facilitating interpretation. The R^2 score measures the fraction of explained variance:

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} \in (-\infty, 1]$$

$R^2 = 1$: perfect prediction; $R^2 = 0$: model is no better than the mean; $R^2 < 0$: model is worse than the mean.

3.2 Binary Cross-Entropy (BCE)

Binary Cross-Entropy is the loss function for binary classification, derived from maximum likelihood under a Bernoulli distribution.

BCE Definition

For N samples with labels $y_i \in \{0, 1\}$ and predicted probabilities $\hat{p}_i = \sigma(z_i) \in (0, 1)$:

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{N} \sum_{i=1}^N [y_i \log \hat{p}_i + (1 - y_i) \log(1 - \hat{p}_i)]$$

Derivation from Bernoulli likelihood. Each observation $y_i \in \{0, 1\}$ follows $y_i \sim \text{Bernoulli}(\hat{p}_i)$. The probability of observing y_i is:

$$p(y_i | \hat{p}_i) = \hat{p}_i^{y_i} (1 - \hat{p}_i)^{1-y_i}$$

The negative log-likelihood over the entire dataset is exactly the BCE.

BCE + Sigmoid gradient. Combining BCE and the sigmoid activation in the output layer, the derivative with respect to the logit z simplifies elegantly:

$$\frac{\partial \mathcal{L}_{\text{BCE}}}{\partial z} = \hat{p} - y$$

This simple form (direct residual) is why sigmoid + BCE is the standard combination for binary classification.

EXAMPLE — Numerical Computation with BCE

Three examples: $(y_1 = 1, \hat{p}_1 = 0.9)$, $(y_2 = 0, \hat{p}_2 = 0.2)$, $(y_3 = 1, \hat{p}_3 = 0.3)$:

$$\mathcal{L}_1 = -\log(0.9) = 0.105 \quad (\text{good prediction})$$

$$\mathcal{L}_2 = -\log(1 - 0.2) = -\log(0.8) = 0.223 \quad (\text{good prediction})$$

$$\mathcal{L}_3 = -\log(0.3) = 1.204 \quad (\text{wrong prediction, high penalty})$$

$$\mathcal{L}_{\text{BCE}} = (0.105 + 0.223 + 1.204)/3 = 0.511$$

Numerical stability. In practice, `F.binary_cross_entropy_with_logits` (PyTorch) is used, which computes:

$$\mathcal{L} = \max(z, 0) - zy + \log(1 + e^{-|z|})$$

This implementation avoids overflow/underflow by computing $\log(\sigma(z))$ without explicitly materialising $\sigma(z)$.

BCE vs. Hinge Loss. SVMs use the hinge loss $\max(0, 1 - y \cdot z)$. BCE is preferred in neural networks because it produces non-zero gradients even far from the margin, accelerating training.

3.3 Softmax and Categorical Cross-Entropy

For K -class classification with one-hot target $\mathbf{y} \in \{0, 1\}^K$, $\sum_k y_k = 1$, and softmax output $\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$:

$$\mathcal{L}_{\text{CCE}} = -\sum_{k=1}^K y_k \log \hat{y}_k = -\log \hat{y}_c$$

where c is the true class (since $y_k = 0$ for $k \neq c$).

Gradient of CCE through softmax. This requires care because $\hat{y}_k$ depends on all z_j .

Step 1. $\frac{\partial \hat{y}_k}{\partial z_j}$. Two cases:

$$\begin{aligned} k = j : \quad & \frac{\partial \hat{y}_k}{\partial z_k} = \hat{y}_k(1 - \hat{y}_k) \\ k \neq j : \quad & \frac{\partial \hat{y}_k}{\partial z_j} = -\hat{y}_k \hat{y}_j \end{aligned}$$

Both cases: $\frac{\partial \hat{y}_k}{\partial z_j} = \hat{y}_k(\mathbf{1}_{k=j} - \hat{y}_j)$.

Step 2. Chain rule:

$$\frac{\partial \mathcal{L}_{\text{CCE}}}{\partial z_j} = \sum_{k=1}^K \frac{\partial \mathcal{L}_{\text{CCE}}}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial z_j} = \sum_{k=1}^K \left(-\frac{y_k}{\hat{y}_k} \right) \hat{y}_k(\mathbf{1}_{k=j} - \hat{y}_j)$$

$$= \sum_{k=1}^K (-y_k)(\mathbf{1}_{k=j} - \hat{y}_j) = -y_j + \hat{y}_j \sum_{k=1}^K y_k = -y_j + \hat{y}_j$$

$$\frac{\partial \mathcal{L}_{\text{CCE}}}{\partial z_j} = \hat{y}_j - y_j$$

Again the gradient is simply prediction minus target — this is not a coincidence but a consequence of the exponential family structure.

EXAMPLE — BCE loss and gradient at $\hat{y} = 0.4934$, $y = 1$

$$\mathcal{L} = -(1) \log(0.4934) - (0) \log(0.5066) = -\log(0.4934) \approx 0.7064$$

$$\frac{\partial \mathcal{L}}{\partial z} = \hat{y} - y = 0.4934 - 1 = -0.5066$$

3.4 Huber Loss and Robust Regression

MSE penalises large errors quadratically, making it sensitive to outliers. L1 loss $|\hat{y} - y|$ is robust but has a non-differentiable kink at 0. The Huber loss combines both:

$$L_\delta(\hat{y}, y) = \begin{cases} \frac{1}{2}(\hat{y} - y)^2 & \text{if } |\hat{y} - y| \leq \delta \\ \delta(|\hat{y} - y| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

For $|r| \leq \delta$ (small errors): quadratic — same as MSE, smooth gradient. For $|r| > \delta$ (large errors): linear in $|r|$ — bounded gradient, robust to outliers.

Gradient.

$$\frac{\partial L_\delta}{\partial \hat{y}} = \begin{cases} \hat{y} - y & \text{if } |\hat{y} - y| \leq \delta \\ \delta \cdot \text{sign}(\hat{y} - y) & \text{otherwise} \end{cases}$$

The gradient is bounded by δ for large residuals, preventing outlier examples from dominating the parameter update.

EXAMPLE — Huber vs. MSE Gradient for an Outlier

$\hat{y} = 1$, $y = 10$ (outlier with $r = -9$), $\delta = 1$.

MSE gradient: $\hat{y} - y = -9$. Update: $\theta \leftarrow \theta + 9\eta$.

Huber gradient: $-\delta \cdot \text{sign}(r) = -1 \cdot (-1) = 1$. Update: $\theta \leftarrow \theta - 1\eta$.

The MSE update is $9\times$ larger, potentially destabilising training. Huber clips the contribution to at most δ .

Choice of δ . Set δ to the expected scale of typical residuals (e.g., the standard deviation of in-distribution residuals). For reward modelling in reinforcement learning from human feedback (RLHF), $\delta = 10$ is common. Huber loss is also used in object detection (smooth L1 for bounding box regression in Faster R-CNN).

MSE is sensitive to outliers. The Huber loss interpolates between MSE and MAE:

$$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{\delta}{2}) & \text{otherwise} \end{cases}$$

Gradient: $\partial L_\delta / \partial \hat{y} = (\hat{y} - y)$ for $|y - \hat{y}| \leq \delta$; $\delta \cdot \text{sign}(\hat{y} - y)$ otherwise. Typical choice: $\delta = 1$.

3.5 Label Smoothing

Replace one-hot target y_k with $\tilde{y}_k = (1 - \epsilon)y_k + \epsilon/K$ ($\epsilon \approx 0.1$). This prevents logits from growing to $\pm\infty$ and improves calibration. The modified loss:

$$\mathcal{L}_{\text{LS}} = -(1 - \epsilon) \log \hat{y}_c - \frac{\epsilon}{K} \sum_{k=1}^K \log \hat{y}_k$$

The second term encourages prediction entropy — a regulariser penalising overconfident distributions.

Gradient analysis. The softmax outputs are $\hat{y}_k = e^{z_k} / \sum_j e^{z_j}$. The gradient of $\mathcal{L}_{\text{LS}}$ w.r.t. logit z_k is:

$$\frac{\partial \mathcal{L}_{\text{LS}}}{\partial z_k} = \hat{y}_k - \tilde{y}_k = \hat{y}_k - \left[(1 - \epsilon)y_k + \frac{\epsilon}{K} \right]$$

For the correct class ($k = c$): gradient = $\hat{y}_c - (1 - \epsilon + \epsilon/K)$. Zero when $\hat{y}_c^* = 1 - \epsilon + \epsilon/K < 1$. For incorrect classes ($k \neq c$): gradient = $\hat{y}_k - \epsilon/K$. Zero when $\hat{y}_k^* = \epsilon/K > 0$.

The uniform floor ϵ/K ensures the model always assigns some probability mass to every class, preventing logit saturation. At the optimum, logit gaps are finite:

$$z_c^* - z_k^* = \log \frac{\hat{y}_c^*}{\hat{y}_k^*} = \log \frac{1 - \epsilon + \epsilon/K}{\epsilon/K} \approx \log \frac{K(1 - \epsilon)}{\epsilon}$$

For $K = 1000$, $\epsilon = 0.1$: optimal gap $\approx \log(9000) \approx 9$. Without label smoothing, the optimal gap is $+\infty$ and logits diverge during training.

Calibration. A model is *well-calibrated* if, among all inputs where it predicts class c with confidence p , approximately fraction p are indeed class c . Hard targets drive confidence to 1.0, causing overconfidence. Label smoothing reduces Expected Calibration Error (ECE):

$$\text{ECE} = \sum_{m=1}^M \frac{|B_m|}{N} |\text{acc}(B_m) - \text{conf}(B_m)|$$

where B_m are confidence bins. Müller et al. (2019) showed label smoothing encourages tight intra-class clustering in penultimate-layer representations, making it useful as a teacher in knowledge distillation — though the smoothed teacher's soft targets are more informative than hard labels.

Method	Top-1 Acc.	ECE	Optimal $\hat{y}_c^*$
Hard labels	76.1%	4.3%	1.000
Label smoothing $\epsilon = 0.1$	77.4%	1.9%	0.9001
Temperature scaling (post-hoc)	76.1%	1.2%	varies

Label smoothing is standard in Inception-v3, ViT, BERT, and most modern classifiers with $\epsilon \in \{0.05, 0.1\}$.

3.6 Focal Loss for Class Imbalance

In object detection, a single image contains $\sim 10^4$ – 10^5 candidate regions (anchors), but only ~ 10 – 50 contain objects. The class imbalance (background vs. foreground) is extreme: $\sim 1000 : 1$. Standard BCE sums over all candidates; easy background examples numerically dominate the gradient.

Focal loss (Lin et al., 2017) reshapes BCE by a modulating factor:

$$\text{FL}(p_t) = -(1 - p_t)^\gamma \log(p_t)$$

where $p_t = \hat{y}$ if $y = 1$, else $p_t = 1 - \hat{y}$, and $\gamma \geq 0$ is the focusing parameter.

Effect of γ .

- $\gamma = 0$: FL = standard BCE.
- Easy example ($p_t = 0.9$): $(1 - 0.9)^2 \log(0.9) = 0.01 \times 0.105 = 0.001$ (down-weighted $100\times$ vs. BCE).
- Hard example ($p_t = 0.1$): $(1 - 0.1)^2 \log(0.1) = 0.81 \times 2.303 = 1.865$ (close to original BCE).

For $\gamma = 2$, the loss is effectively dominated by examples with $p_t < 0.5$ — the hard, uncertain predictions.

p_t	BCE	FL ($\gamma = 2$)	Ratio FL/BCE
0.9	0.105	0.001	$0.01\times$
0.7	0.357	0.032	$0.09\times$
0.5	0.693	0.173	$0.25\times$
0.3	1.204	0.588	$0.49\times$
0.1	2.303	1.865	$0.81\times$

In RetinaNet (Lin et al., 2017), focal loss with $\gamma = 2$ and $\alpha = 0.25$ (class weighting) achieves accuracy matching two-stage detectors (Faster R-CNN) with a simpler single-stage architecture.

3.7 Surrogate Losses and Zero-One Loss

The ideal objective for classification is the **zero-one loss**:

$$\mathcal{L}_{0/1} = \frac{1}{N} \sum_{i=1}^N \mathbf{1}[\hat{y}_i \neq y_i]$$

This counts misclassified examples directly. Unfortunately it is non-convex, discontinuous, and has gradient zero almost everywhere — making it unsuitable for gradient-based optimisation.

Convex surrogates. We replace $\mathcal{L}_{0/1}$ with a convex upper bound that is differentiable:

Surrogate	Formula	Gradient	Application
Hinge	$\max(0, 1 - y \cdot z)$	$-y \mathbf{1}[yz < 1]$	SVM
Logistic/BCE	$\log(1 + e^{-yz})$	$-y\sigma(-yz)$	Logistic regression
Exponential	e^{-yz}	$-ye^{-yz}$	AdaBoost
Squared hinge	$\max(0, 1 - yz)^2$	$-2y(1 - yz)^+$	LS-SVM

(Here $y \in \{-1, +1\}$ and z is the logit.)

Calibration. A loss is *calibrated* if its minimiser yields the Bayes-optimal posterior $P(y = 1|x)$. The logistic loss is calibrated; the hinge loss is not (it does not recover class probabilities).

3.8 Numerical Example: Loss Surfaces and Saddle Points

Consider a two-parameter quadratic loss $\mathcal{L}(\theta_1, \theta_2) = \theta_1^2 - \theta_2^2$ (a saddle at the origin).

θ_1	θ_2	$\mathcal{L}$
0	0	0 (saddle)
1	0	1
0	1	-1
1	1	0

Gradient: $\nabla \mathcal{L} = (2\theta_1, -2\theta_2)^\top$. At the origin, $\nabla \mathcal{L} = \mathbf{0}$ but the Hessian $\mathbf{H} = \text{diag}(2, -2)$ is indefinite — a saddle point, not a minimum. Gradient descent escapes saddles via noise in stochastic gradients; second-order methods detect them via the Hessian's negative eigenvalues.

Deep network loss landscapes. Goodfellow et al. (2015) showed empirically that the loss along a linear interpolation between two trained checkpoints is nearly monotone — the landscape is benign along certain paths. Choromanska et al. (2015) connected the loss surface of deep ReLU nets to the Hamiltonian of a spin glass, showing that local minima cluster near the global minimum for large networks.

3.9 Contrastive and Ranking Losses

Beyond prediction losses (MSE, BCE, CCE), a family of losses is designed to learn *similarity structure* in embedding space. These are central to self-supervised and metric learning systems.

Contrastive loss (Hadsell et al., 2006). Given a pair of examples $(\mathbf{x}_i, \mathbf{x}_j)$ with binary label $y_{ij} = 1$ (similar) or $y_{ij} = 0$ (dissimilar), and embeddings $\mathbf{z}_i = f(\mathbf{x}_i)$, $\mathbf{z}_j = f(\mathbf{x}_j)$:

$$\mathcal{L}_{\text{contrastive}} = y_{ij} D_{ij}^2 + (1 - y_{ij}) \max(0, m - D_{ij})^2$$

where $D_{ij} = \|\mathbf{z}_i - \mathbf{z}_j\|_2$ and $m > 0$ is a margin. The loss pushes similar pairs together ($D_{ij} \rightarrow 0$) and repels dissimilar pairs beyond margin m ($D_{ij} > m$).

Triplet loss (Schroff et al., FaceNet, 2015). Given an anchor $\mathbf{x}^a$, positive $\mathbf{x}^p$ (same class), and negative $\mathbf{x}^n$ (different class):

$$\mathcal{L}_{\text{triplet}} = \max(0, \|\mathbf{z}^a - \mathbf{z}^p\|_2^2 - \|\mathbf{z}^a - \mathbf{z}^n\|_2^2 + \alpha)$$

where $\alpha > 0$ is the margin. The loss is zero when the anchor is at least α closer to the positive than to the negative. This directly optimises the embedding geometry: a classifier trained on these embeddings will achieve low error even on unseen classes (few-shot learning).

InfoNCE / NT-Xent (Contrastive Self-Supervised Learning). Given a batch of N examples, each augmented twice to produce pairs $\{(\mathbf{x}_i, \mathbf{x}'_i)\}_{i=1}^N$. For example i , the $2N - 2$ other augmented views are treated as negatives:

$$\mathcal{L}_{\text{NCE}} = -\frac{1}{N} \sum_{i=1}^N \log \frac{\exp(\mathbf{z}_i^\top \mathbf{z}'_i / \tau)}{\sum_{j \neq i} \exp(\mathbf{z}_i^\top \mathbf{z}_j / \tau)}$$

where $\tau > 0$ is a temperature hyperparameter and $\mathbf{z}$ are ℓ_2 -normalised embeddings. This is the loss used in SimCLR (Chen et al., 2020) and MoCo.

The InfoNCE loss can be interpreted as maximising a lower bound on the mutual information $I(\mathbf{z}_i; \mathbf{z}'_i)$:

$$I(\mathbf{z}_i; \mathbf{z}'_i) \geq \log N - \mathcal{L}_{\text{NCE}}$$

A lower temperature τ creates sharper distributions (harder negatives weighted more), leading to better representations but requiring larger batches for stable negatives. SimCLR uses $\tau = 0.07$ and batch size $B = 4096$.

EXAMPLE — InfoNCE Gradient Intuition

For a single anchor $\mathbf{z}_i$, the gradient of $\mathcal{L}_{\text{NCE}}$ w.r.t. $\mathbf{z}_i$ is:

$$\frac{\partial \mathcal{L}_{\text{NCE}}}{\partial \mathbf{z}_i} = -\frac{1}{\tau} \left[\mathbf{z}'_i - \sum_j p_{ij} \mathbf{z}_j \right]$$

where $p_{ij} = \text{softmax}(\mathbf{z}_i^\top \mathbf{z}_j / \tau)$. The gradient pushes $\mathbf{z}_i$ toward its positive $\mathbf{z}'_i$ and away from the negative-weighted centroid $\sum_j p_{ij} \mathbf{z}_j$. Hard negatives (those with high similarity p_{ij}) receive the largest repulsion — the loss automatically mines hard negatives via the softmax weighting.

Backpropagation

We derive the complete backpropagation algorithm from the chain rule applied to a computational graph. Every gradient is computed symbolically and then verified numerically on the 3-2-1 network from Part 2.

4.1 The Chain Rule on Computational Graphs

Backpropagation is the systematic application of the *chain rule* on a computational graph. Understanding its structure is the prerequisite for correctly implementing any custom layer.

Chain rule — scalar case.

$$\frac{d\mathcal{L}}{dx} = \frac{d\mathcal{L}}{dz} \cdot \frac{dz}{dx}$$

If $z = f(x)$ and $\mathcal{L} = g(z)$, the derivative of $\mathcal{L}$ with respect to x is the product of the local derivatives.

Chain rule — vector case.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \left(\frac{\partial \mathbf{z}}{\partial \mathbf{x}} \right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{z}}$$

where $\partial \mathbf{z} / \partial \mathbf{x}$ is the Jacobian $J \in \mathbb{R}^{m \times n}$ (with $\mathbf{z} \in \mathbb{R}^m$, $\mathbf{x} \in \mathbb{R}^n$). Note the transpose: backpropagation multiplies transposed Jacobians.

Computational graph. A neural network defines a DAG (Directed Acyclic Graph) where:

- Every leaf node is a parameter or an input.
- Every internal node is a differentiable operation (add, matmul, relu, log, ...).
- The graph output is the loss $\mathcal{L}$.

Backpropagation traverses the graph in reverse (from the output node to the leaf nodes), accumulating gradients via the chain rule.

EXAMPLE — Computational Graph for a Linear Layer

Forward: $\mathcal{L} = g(\mathbf{a}) = g(\sigma(\mathbf{z})) = g(\sigma(\mathbf{W}\mathbf{x} + \mathbf{b}))$

Backward, step by step:

1. $\delta 0_a = \partial \mathcal{L} / \partial \mathbf{a}$ (given from upstream)
2. $\delta 0_z = \delta 0_a \odot \sigma'(\mathbf{z})$
3. $\partial \mathcal{L} / \partial \mathbf{W} = \delta 0_z \mathbf{x}^\top$
4. $\partial \mathcal{L} / \partial \mathbf{b} = \delta 0_z$
5. $\partial \mathcal{L} / \partial \mathbf{x} = \mathbf{W}^\top \delta 0_z$ (passed to the previous layer)

Forward vs. reverse mode.

- **Forward mode AD:** propagates derivatives with respect to one input at a time. Efficient when $|\text{inputs}| \ll |\text{outputs}|$.
- **Reverse mode AD (backpropagation):** propagates gradients from output to all inputs in a single pass. Efficient for a scalar $\mathcal{L}$ (such as the loss), regardless of the number of parameters. Complexity: $O(|\text{forward}|)$.

For networks with $P \sim 10^9$ parameters and a scalar loss, reverse mode is the only practical option.

Local Gradient and Global Gradient

For each node v in the graph:

$$\underbrace{\frac{\partial \mathcal{L}}{\partial v}}_{|\{z\}| \text{ global gradient}} = \underbrace{\frac{\partial \mathcal{L}}{\partial u}}_{|\{z\}| \text{ global gradient of parent}} \cdot \underbrace{\frac{\partial u}{\partial v}}_{|\{z\}| \text{ local gradient}}$$

The local gradient depends only on the local operation; the global gradient accumulates information from all upstream nodes.

4.2 Complete Algorithm

Define the **error signal** at layer l as:

$$\square^{(l)} \triangleq \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}} \in \mathbb{R}^{n_l}$$

Algorithm: Backpropagation

Forward pass: compute and store $\mathbf{z}^{(l)}, \mathbf{a}^{(l)}$ for all l .

Output layer error:

$$\square^{(L)} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \odot \phi'^{(L)}(\mathbf{z}^{(L)})$$

(with BCE+sigmoid: $\square^{(L)} = \hat{\mathbf{y}} - \mathbf{y}$)

Backpropagate: for $l = L - 1, L - 2, \dots, 1$:

$$\square^{(l)} = (\mathbf{W}^{(l+1)\top} \square^{(l+1)}) \odot \phi'^{(l)}(\mathbf{z}^{(l)})$$

Parameter gradients:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \square^{(l)} (\mathbf{a}^{(l-1)})^\top, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \square^{(l)}$$

Derivation of the recursion. We derive $\square^{(l)}$ from $\square^{(l+1)}$.

By definition, $z_j^{(l)}$ affects $\mathcal{L}$ only through $a_j^{(l)} = \phi(z_j^{(l)})$, which then feeds into every $z_i^{(l+1)} = \sum_j W_{ij}^{(l+1)} a_j^{(l)} + b_i^{(l+1)}$. So:

$$\begin{aligned} \delta_j^{(l)} &= \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} = \frac{\partial \mathcal{L}}{\partial a_j^{(l)}} \cdot \phi'(z_j^{(l)}) \\ \frac{\partial \mathcal{L}}{\partial a_j^{(l)}} &= \sum_{i=1}^{n_{l+1}} \frac{\partial \mathcal{L}}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial a_j^{(l)}} = \sum_{i=1}^{n_{l+1}} \delta_i^{(l+1)} \cdot W_{ij}^{(l+1)} \end{aligned}$$

In vector form: $\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(l)}} = \mathbf{W}^{(l+1)\top} \square^{(l+1)}$. Therefore:

$$\square^{(l)} = (\mathbf{W}^{(l+1)\top} \square^{(l+1)}) \odot \phi'(\mathbf{z}^{(l)})$$

Derivation of weight gradient.

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_i^{(l)}} \cdot \frac{\partial z_i^{(l)}}{\partial W_{ij}^{(l)}} = \delta_i^{(l)} \cdot a_j^{(l-1)}$$

Stacking all (i, j) pairs: $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)} (\mathbf{a}^{(l-1)})^\top$.

4.3 Complete Numerical Example — Backpropagation

We continue the 3-2-1 network from Part 2 with target $y = 1$, and then verify using a numerical gradient check.

Numerical gradient check. For any parameter W_{ij} , the numerical gradient is:

$$\frac{\partial \mathcal{L}}{\partial W_{ij}} \approx \frac{\mathcal{L}(W_{ij} + h) - \mathcal{L}(W_{ij} - h)}{2h}, \quad h = 10^{-5}$$

The *relative error* between analytical and numerical gradients should be $< 10^{-5}$ for correct backpropagation implementation. This is the standard debugging tool for new architectures.

From the forward pass: $\mathbf{z}^{(1)} = (0.75, -0.5)^\top$, $\mathbf{a}^{(1)} = (0.6792, 0.3775)^\top$, $z^{(2)} = -0.0266$, $\hat{y} = 0.4934$.

Output layer ($l = 2$), BCE+sigmoid:

$$\delta^{(2)} = \hat{y} - y = 0.4934 - 1.0 = -0.5066$$

Gradients for $\mathbf{W}^{(2)}$ and $\mathbf{b}^{(2)}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} = \delta^{(2)} \cdot (\mathbf{a}^{(1)})^\top = (-0.5066) \cdot (0.6792 \quad 0.3775) = (-0.3441 \quad -0.1912)$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(2)}} = -0.5066$$

Propagate to layer 1:

$$\begin{aligned} \mathbf{W}^{(2)\top} \delta^{(2)} &= \begin{pmatrix} 0.7 \\ -0.8 \end{pmatrix} (-0.5066) = \begin{pmatrix} -0.3546 \\ 0.4053 \end{pmatrix} \\ \phi'(\mathbf{z}^{(1)}) &= \begin{pmatrix} \sigma'(0.75) \\ \sigma'(-0.5) \end{pmatrix} = \begin{pmatrix} 0.6792(1 - 0.6792) \\ 0.3775(1 - 0.3775) \end{pmatrix} = \begin{pmatrix} 0.2179 \\ 0.2350 \end{pmatrix} \\ \boldsymbol{\delta}^{(1)} &= \begin{pmatrix} -0.3546 \\ 0.4053 \end{pmatrix} \odot \begin{pmatrix} 0.2179 \\ 0.2350 \end{pmatrix} = \begin{pmatrix} -0.0773 \\ 0.0952 \end{pmatrix} \end{aligned}$$

Gradients for $\mathbf{W}^{(1)}$ and $\mathbf{b}^{(1)}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} = \boldsymbol{\delta}^{(1)} (\mathbf{x})^\top = \begin{pmatrix} -0.0773 \\ 0.0952 \end{pmatrix} (1.0, 0.5, -1.0) = \begin{pmatrix} -0.0773 & -0.0387 & 0.0773 \\ 0.0952 & 0.0476 & -0.0952 \end{pmatrix}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}} = \begin{pmatrix} -0.0773 \\ 0.0952 \end{pmatrix}$$

All gradients have been computed — exactly what gradient descent needs.

4.4 Vector-Jacobian Products and Efficient Backpropagation

The core operation in reverse-mode AD is the **vector-Jacobian product** (VJP):

$$\bar{\mathbf{x}} = \bar{\mathbf{y}}^\top J_f(\mathbf{x})$$

where $\bar{\mathbf{y}} = \partial\mathcal{L}/\partial\mathbf{y}$ is the upstream gradient and $J_f \in \mathbb{R}^{m \times n}$ is the Jacobian of $\mathbf{y} = f(\mathbf{x})$. We never explicitly form J_f ; instead we implement a `vjp` function that computes $\bar{\mathbf{x}}$ directly.

VJP for Common Operations

- **Linear layer:** $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$. $J_f = \mathbf{W}$. VJP: $\bar{\mathbf{x}} = \mathbf{W}^\top \bar{\mathbf{y}}$, $\bar{\mathbf{W}} = \bar{\mathbf{y}}\mathbf{x}^\top$, $\bar{\mathbf{b}} = \bar{\mathbf{y}}$.
- **Element-wise activation:** $\mathbf{y} = \phi(\mathbf{x})$. $J_f = \text{diag}(\phi'(\mathbf{x}))$. VJP: $\bar{\mathbf{x}} = \bar{\mathbf{y}} \odot \phi'(\mathbf{x})$.
- **Softmax:** $\mathbf{y} = \text{softmax}(\mathbf{x})$. $J_f = \text{diag}(\mathbf{y}) - \mathbf{y}\mathbf{y}^\top$. VJP: $\bar{\mathbf{x}} = \mathbf{y} \odot (\bar{\mathbf{y}} - \bar{\mathbf{y}}^\top \mathbf{y} \mathbf{1})$.
- **Reshape/transpose:** VJP is the inverse reshape/transpose of $\bar{\mathbf{y}}$.

Derivation of the linear layer VJP. For $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$, we need $\bar{\mathbf{W}} = \partial\mathcal{L}/\partial\mathbf{W}$. Using the chain rule:

$$\delta\mathcal{L} = \bar{\mathbf{y}}^\top \delta\mathbf{y} = \bar{\mathbf{y}}^\top (\delta\mathbf{W})\mathbf{x} = \text{tr}(\bar{\mathbf{y}}^\top (\delta\mathbf{W})\mathbf{x}) = \text{tr}(\mathbf{x}\bar{\mathbf{y}}^\top \delta\mathbf{W})$$

Since $\delta\mathcal{L} = \text{tr}((\bar{\mathbf{W}})^\top \delta\mathbf{W})$ by definition of the matrix gradient, we have $\bar{\mathbf{W}} = \bar{\mathbf{y}}\mathbf{x}^\top \in \mathbb{R}^{m \times n}$ — an outer product between the upstream gradient and the input. This is the fundamental formula of backpropagation: **the weight gradient is an outer product of the upstream error and the layer input**.

Batched VJP. With batch size B , $\mathbf{X} \in \mathbb{R}^{B \times n}$ and $\mathbf{Y} = \mathbf{X}\mathbf{W}^\top \in \mathbb{R}^{B \times m}$. The weight gradient accumulates over the batch:

$$\bar{\mathbf{W}} = \bar{\mathbf{Y}}^\top \mathbf{X} \in \mathbb{R}^{m \times n}$$

This is a single matrix multiplication — explaining why backpropagation on a batch is only marginally more expensive than on a single example.

Higher-order gradients. VJPs compose: to compute second-order gradients (Hessian-vector products $H\mathbf{v}$), differentiate the VJP operation itself. In JAX: `jvp(grad(f), (x,), (v,))` computes $H(\mathbf{x})\mathbf{v}$ in $O(\text{forward})$ time. This is used in natural gradient methods, meta-learning (MAML), and conjugate gradient second-order optimisers.

Computational cost of VJPs. Each VJP costs $O(\text{forward pass})$. The total backward pass cost for an L -layer network is $O(L \times \text{layer forward cost})$ — which is why backpropagation is only 2–3× more expensive than a single forward pass, regardless of P (number of parameters).

4.5 Backpropagation Through Non-Smooth Operations

ReLU is not differentiable at $x = 0$. In practice, frameworks define the subgradient $\partial\text{ReLU}(0) = 0$ (or sometimes $1/2$). This works because:

- The set $\{x = 0\}$ has measure zero; the probability of a floating-point value being exactly zero is negligible.
- Subgradient descent on non-smooth convex functions converges, and empirically works for non-convex networks too.

Subgradients. A vector $\mathbf{g}$ is a subgradient of f at $\mathbf{x}$ if for all $\mathbf{y}$:

$$f(\mathbf{y}) \geq f(\mathbf{x}) + \mathbf{g}^\top (\mathbf{y} - \mathbf{x})$$

The subdifferential $\partial f(\mathbf{x})$ is the set of all subgradients. For ReLU:

$$\partial \text{ReLU}(x) = \begin{cases} \{1\} & x > 0 \\ [0, 1] & x = 0 \\ \{0\} & x < 0 \end{cases}$$

Any element of the subdifferential (e.g. 0, 1, or 1/2 at $x = 0$) is a valid subgradient for the backward pass. PyTorch uses 0 at $x = 0$ for ReLU; some frameworks use 1/2.

Dead neurons. With the 0 convention at $x = 0$, a neuron that receives large negative pre-activations $z < 0$ during training gets zero gradient and its weights are never updated — a “dead” ReLU. Prevention: proper initialisation (He), leaky ReLU ($\text{LeakyReLU}(x) = \max(0.01x, x)$), or GELU/Swish which have nonzero gradients everywhere.

Straight-Through Estimator (STE). For discrete operations (quantisation, binarisation) the gradient is zero almost everywhere. The STE treats the forward pass as a hard threshold but the backward pass as the identity:

$$\frac{\partial}{\partial x} \text{round}(x) \approx 1 \quad (\text{STE})$$

More generally, the STE replaces a non-differentiable forward function f with a differentiable surrogate $\hat{f}$ in the backward pass:

$$\text{Forward: } y = f(x), \quad \text{Backward: } \bar{x} = \bar{y} \cdot \hat{f}'(x)$$

For quantisation-aware training, $\hat{f}'(x) = 1$ for $x \in [-1, 1]$ and 0 outside (clipped STE). Despite being a biased estimator, it works well in practice and is used in training quantised neural networks (QNNs) and binary neural networks.

EXAMPLE — Backprop Through MaxPool

MaxPool with window 2×2 on a 4×1 input $\mathbf{x} = (3, 1, 4, 2)^\top$: output $\mathbf{y} = (3, 4)^\top$ (indices $\{0, 2\}$ are the maxima).

Upstream gradient $\bar{\mathbf{y}} = (0.5, -0.3)^\top$.

VJP: route each upstream gradient to the argmax position, zero elsewhere.

$$\bar{\mathbf{x}} = (0.5, 0, -0.3, 0)^\top$$

Non-max positions receive zero gradient (their output did not affect the loss). This is why max-pooling can cause “dead” gradient paths; average-pooling distributes gradients more uniformly.

Optimization Algorithms

We derive every parameter update rule used in practice: vanilla gradient descent, momentum, RMSProp, and Adam. We show why each variant improves on the previous.

5.1 Gradient Descent and Variants

The optimisation of a neural network's parameters proceeds via variants of gradient descent, which iteratively update the weights in the direction of steepest descent of the loss.

Batch Gradient Descent

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} \mathcal{L}(\theta_t)$$

where $\eta > 0$ is the *learning rate* and $\nabla_{\theta} \mathcal{L} = \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} \ell_i(\theta)$ is the gradient over the entire dataset.

Stochastic Gradient Descent (SGD). With large datasets, computing the exact gradient is prohibitive. SGD uses a single sample (x_i, y_i) per step:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} \ell_i(\theta_t)$$

The gradient is a *noisy estimator* of the true one: $\mathbb{E}[\nabla \ell_i] = \nabla \mathcal{L}$. The noise, paradoxically, can help generalisation (escaping flat minima, implicit regularisation).

Mini-batch SGD. A practical compromise: batches of $B = 32, 64, 128, 256$ samples.

$$\nabla_{\theta} \mathcal{L}_{\mathcal{B}} = \frac{1}{B} \sum_{i \in \mathcal{B}} \nabla_{\theta} \ell_i(\theta)$$

The gradient variance scales as $\text{Var}[\nabla \ell_i]/B$: larger batches give more accurate estimates but often worsen generalisation (Keskar et al., 2017).

EXAMPLE — One SGD Iteration

Scalar parameter θ , loss $\mathcal{L}(\theta) = \theta^2$, $\eta = 0.1$, $\theta_0 = 3.0$:

$$t = 0 : \quad \nabla \mathcal{L} = 2\theta = 6.0, \quad \theta_1 = 3.0 - 0.1 \cdot 6.0 = 2.4$$

$$t = 1 : \quad \nabla \mathcal{L} = 4.8, \quad \theta_2 = 2.4 - 0.48 = 1.92$$

$$t = 2 : \quad \nabla \mathcal{L} = 3.84, \quad \theta_3 = 1.92 - 0.384 = 1.536$$

Convergence is geometric: $\theta_t = \theta_0(1 - 2\eta)^t = 3.0 \cdot 0.8^t$. Converges to 0 for $|1 - 2\eta| < 1$, i.e., $0 < \eta < 1$.

Momentum (SGD+M).

$$\mathbf{v}_{t+1} = \mu \mathbf{v}_t - \eta \nabla_{\theta} \mathcal{L}_t, \quad \theta_{t+1} = \theta_t + \mathbf{v}_{t+1}$$

The momentum term $\mu \in [0.9, 0.99]$ accumulates an exponential moving average of past gradients, dampening oscillations in high-curvature directions and accelerating progress in low-curvature directions. Physical analogy: a ball rolling down a hill gains inertia, passes over small bumps, and slows on flat ground.

Nesterov Accelerated Gradient (NAG).

$$\theta_{t+1} = \theta_t + \mu \mathbf{v}_t - \eta \nabla_{\theta} \mathcal{L}(\theta_t + \mu \mathbf{v}_t)$$

NAG computes the gradient at the “lookahead” point $\theta_t + \mu \mathbf{v}_t$, achieving theoretically faster convergence (Nesterov, 1983): $O(1/t^2)$ vs $O(1/t)$ for standard GD on convex functions.

Convergence proof for quadratic loss. Consider $\mathcal{L}(\theta) = \frac{1}{2}a\theta^2$ (strongly convex with $\mu = a$, L -smooth with $L = a$). With learning rate $\eta = 1/a$:

$$\theta_{t+1} = \theta_t - \frac{1}{a} \cdot a\theta_t = \theta_t(1 - 1) = 0$$

Convergence in 1 step! For general η :

$$\theta_{t+1} = \theta_t - \eta a \theta_t = (1 - \eta a) \theta_t$$

$$\theta_t = (1 - \eta a)^t \theta_0$$

Convergence requires $|1 - \eta a| < 1$, i.e. $0 < \eta < 2/a$. Optimal $\eta = 1/a$ gives $\theta_1 = 0$.

For a general quadratic $\mathcal{L}(\square) = \frac{1}{2}\square^\top \mathbf{A} \square$ with eigenvalues $\lambda_{\min} = \mu$, $\lambda_{\max} = L$:

$$\|\square_t - \square^*\|^2 \leq \left(\frac{\kappa - 1}{\kappa + 1} \right)^{2t} \|\square_0 - \square^*\|^2, \quad \kappa = L/\mu$$

with optimal fixed step size $\eta^* = 2/(\mu + L)$. NAG improves the contraction factor to $(\sqrt{\kappa} - 1)/(\sqrt{\kappa} + 1)$ — the square root of GD’s factor, meaning quadratically faster.

EXAMPLE — GD vs. NAG on Ill-Conditioned Quadratic

$\mathcal{L}(\theta_1, \theta_2) = 50\theta_1^2 + 0.5\theta_2^2$, so $\kappa = 50/0.5 = 100$, $\eta^* = 2/50.5 \approx 0.0396$.

GD contraction factor: $(100 - 1)/(100 + 1) \approx 0.980$ per step. Steps to halve error: $\log(0.5)/\log(0.980) \approx 34$.

NAG contraction factor: $(\sqrt{100} - 1)/(\sqrt{100} + 1) = 9/11 \approx 0.818$ per step. Steps to halve error: $\log(0.5)/\log(0.818) \approx 3.5$.

NAG is approximately 10× faster on this ill-conditioned problem, and the gap grows with κ .

Choosing the learning rate. Too large an η causes divergence (the loss increases); too small causes slow convergence. Rule of thumb: start with $\eta = 0.01$ for SGD, $\eta = 3 \times 10^{-4}$ for Adam, and use a learning rate scheduler (Section 11.2).

5.2 Adam — Every Step Explained

Adam (Adaptive Moment Estimation, Kingma & Ba, 2015) is the most widely used optimiser in modern deep learning. It combines first-order momentum with adaptive estimation of the gradient variance.

Adam Algorithm — Full Form

Given: η (learning rate), $\beta_1 \in [0, 1]$ (first-moment decay, typical 0.9), $\beta_2 \in [0, 1]$ (second-moment decay, typical 0.999), ε (numerical stabilisation, typical 10^{-8}).

Initialisation: $\mathbf{m}_0 = \mathbf{0}$, $\mathbf{v}_0 = \mathbf{0}$, $t = 0$.

For each step $t = 1, 2, \dots$:

1. Compute the gradient: $\mathbf{g}_t = \nabla_{\theta} \mathcal{L}_t(\theta_{t-1})$
2. Update first moment: $\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \mathbf{g}_t$
3. Update second moment: $\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2$

4. Bias correction: $\hat{\mathbf{m}}_t = \mathbf{m}_t / (1 - \beta_1^t)$, $\hat{\mathbf{v}}_t = \mathbf{v}_t / (1 - \beta_2^t)$
5. Update parameters: $\theta_t = \theta_{t-1} - \eta \cdot \hat{\mathbf{m}}_t / (\sqrt{\hat{\mathbf{v}}_t} + \varepsilon)$

All vector operations are element-wise.

Meaning of the two moments.

First moment $\mathbf{m}_t$: exponential moving average of the gradients. Acts like momentum in SGD+M, dampening oscillations.

Second moment $\mathbf{v}_t$: exponential moving average of the squared gradients. Estimates the (un-centred) variance of the gradient for each parameter.

Why the bias correction. At the start of training, $\mathbf{m}_0 = \mathbf{v}_0 = \mathbf{0}$. If $\beta_1 = 0.9$, after the first step $\mathbf{m}_1 = (1 - 0.9)\mathbf{g}_1 = 0.1\mathbf{g}_1$: the first moment is strongly underestimated. The correction $\hat{\mathbf{m}}_1 = \mathbf{m}_1 / (1 - 0.9^1) = \mathbf{g}_1$ removes this initial bias.

EXAMPLE — Adam Step by Step

Scalar parameter θ , gradient at step 1: $g_1 = 0.5$. $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\eta = 0.001$, $\varepsilon = 10^{-8}$.

$$m_1 = 0.9 \cdot 0 + 0.1 \cdot 0.5 = 0.05$$

$$v_1 = 0.999 \cdot 0 + 0.001 \cdot 0.25 = 0.00025$$

$$\hat{m}_1 = 0.05 / (1 - 0.9) = 0.5$$

$$\hat{v}_1 = 0.00025 / (1 - 0.999) = 0.25$$

$$\Delta\theta = -0.001 \cdot \frac{0.5}{\sqrt{0.25} + 10^{-8}} = -0.001 \cdot \frac{0.5}{0.5} = -0.001$$

At the first step, Adam reduces to an update of magnitude η : the effective learning rate is $\approx \eta$ regardless of the gradient magnitude.

Geometric interpretation. The term $\hat{\mathbf{m}}_t / \sqrt{\hat{\mathbf{v}}_t}$ normalises the gradient by its estimated standard deviation. For parameters with historically variable gradients, the effective learning rate $\eta / \sqrt{\hat{v}_t}$ is small (caution); for stable parameters, it is larger. Adam self-adapts per coordinate.

Limitations of Adam.

- It can converge to sub-optimal solutions for certain problems (Reddi et al., 2018).
- Standard weight decay does not work well with Adam (see AdamW, Section 5.3).
- For very large LLMs, the memory for $\mathbf{m}$ and $\mathbf{v}$ (two copies of the parameters) is expensive.

5.3 AdamW and Decoupled Weight Decay

In Adam, L2 regularisation adds $\lambda \square$ to the gradient, which gets divided by $\sqrt{\hat{\mathbf{v}}}$ — the weight decay is non-uniform and scaled by the adaptive rate. **AdamW** decouples this:

$$\square_t = \square_{t-1} - \alpha \left(\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon} + \lambda \square_{t-1} \right)$$

The decay $\lambda \square_{t-1}$ is not scaled by the adaptive denominator.

Why L2 regularisation $\neq$ weight decay in Adam. In SGD, $\theta_t = (1 - \eta\lambda)\theta_{t-1} - \eta g_t$: true weight decay multiplies weights by $(1 - \eta\lambda)$ each step. In Adam, adding $\lambda\theta$ to g_t makes it enter $\hat{v}_t = \text{avg}(g_t + \lambda\theta_t)^2$. For a parameter with large gradients $g_t \gg \lambda\theta_t$: $\hat{v}_t$ is large, so the effective step $\alpha/\sqrt{\hat{v}_t}$ is small, and the weight decay contribution is also divided by $\sqrt{\hat{v}_t}$ — *undermining* regularisation for the most active parameters. AdamW applies weight decay uniformly, regardless of gradient magnitude.

Full per-parameter update equations. At step t for a single parameter θ :

$$\begin{aligned} g_t &= \nabla_{\theta} \mathcal{L}_t(\theta_{t-1}) \\ m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\ \hat{m}_t &= m_t / (1 - \beta_1^t) \\ \hat{v}_t &= v_t / (1 - \beta_2^t) \\ \theta_t &= \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} - \alpha \lambda \theta_{t-1} \end{aligned}$$

The final line differs from Adam only in the last term: the weight decay $\alpha \lambda \theta_{t-1}$ is added directly to the parameter update, bypassing the adaptive denominator.

Effective regularisation comparison. Consider a parameter where $|g_t| = 10$ and $\lambda\theta = 0.01$. With $\hat{v}_t \approx 100$ (i.e. $\sqrt{\hat{v}_t} \approx 10$):

- Adam (L2): weight decay contribution to update $= \alpha \lambda \theta / \sqrt{\hat{v}_t} = \alpha \cdot 0.001$ (100 $\times$ attenuated).
- AdamW: weight decay contribution $= \alpha \lambda \theta = \alpha \cdot 0.01$ (full strength).

The attenuation in Adam-L2 is parameter-specific and unpredictable, making it hard to choose a meaningful λ . AdamW restores the intended regularisation strength uniformly.

Hyperparameters. Typical values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, $\alpha \in [10^{-4}, 10^{-3}]$, $\lambda \in \{0.01, 0.1\}$. Embedding layers, LayerNorm parameters, and bias terms are typically excluded from weight decay (set $\lambda = 0$ for those parameter groups), as penalising them is not beneficial and can harm training.

Implementation. In PyTorch, `torch.optim.AdamW` correctly decouples weight decay. The older `torch.optim.Adam` with `weight_decay` applies L2 (coupled) — different behaviour. AdamW is the standard Transformer optimiser (BERT, GPT, LLaMA) and generalises well across tasks.

5.4 Convergence Analysis

L-smooth and μ -strongly convex

f is L -smooth if $\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\| \leq L \|\mathbf{x} - \mathbf{y}\|$ for all $\mathbf{x}, \mathbf{y}$.

f is μ -strongly convex if $f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{y} - \mathbf{x}) + \frac{\mu}{2} \|\mathbf{y} - \mathbf{x}\|^2$.

Gradient descent lemma. L -smoothness implies that each gradient step decreases f by at least:

$$f(\square - \frac{1}{L} \nabla f(\square)) \leq f(\square) - \frac{1}{2L} \|\nabla f(\square)\|^2$$

Proof. By L -smoothness, $f(\mathbf{y}) \leq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{y} - \mathbf{x}) + \frac{L}{2} \|\mathbf{y} - \mathbf{x}\|^2$. Setting $\mathbf{y} = \mathbf{x} - \frac{1}{L} \nabla f(\mathbf{x})$:

$$f(\mathbf{y}) \leq f(\mathbf{x}) - \frac{1}{L} \|\nabla f(\mathbf{x})\|^2 + \frac{L}{2} \cdot \frac{1}{L^2} \|\nabla f(\mathbf{x})\|^2 = f(\mathbf{x}) - \frac{1}{2L} \|\nabla f(\mathbf{x})\|^2 \quad \square$$

Sublinear convergence (convex case). For L -smooth convex f with step size $\alpha = 1/L$:

$$f(\square_T) - f^* \leq \frac{L \|\square_0 - \square^*\|^2}{2T} = O(1/T)$$

Proof sketch. By the descent lemma, $\sum_{t=0}^{T-1} \|\nabla f(\square_t)\|^2 \leq 2L(f(\square_0) - f^*)$. Convexity gives $f(\square_t) - f^* \leq \nabla f(\square_t)^\top (\square_t - \square^*)$. Combining with the update rule and telescoping gives the bound. $\square$

Linear convergence (strongly convex case). For L -smooth μ -strongly convex f :

$$\|\square_T - \square^*\|^2 \leq \left(1 - \frac{\mu}{L}\right)^T \|\square_0 - \square^*\|^2$$

The condition number $\kappa = L/\mu \geq 1$ controls the convergence rate. Large κ (ill-conditioned problem) means slow convergence: to reduce the error by $1/e$, we need $T \geq \kappa$ steps.

Proof. Strong convexity implies $f(\mathbf{x}) \geq f(\square^*) + \frac{\mu}{2} \|\mathbf{x} - \square^*\|^2$. Combining with the descent lemma and the identity:

$$\|\square_{t+1} - \square^*\|^2 = \left\| \square_t - \frac{1}{L} \nabla f(\square_t) - \square^* \right\|^2 = \|\square_t - \square^*\|^2 - \frac{2}{L} \nabla f(\square_t)^\top (\square_t - \square^*) + \frac{1}{L^2} \|\nabla f(\square_t)\|^2$$

Using strong convexity on the middle term: $\nabla f(\square_t)^\top (\square_t - \square^*) \geq f(\square_t) - f^* + \frac{\mu}{2} \|\square_t - \square^*\|^2 \geq \frac{\mu}{2} \|\square_t - \square^*\|^2$. Substituting and simplifying yields the geometric contraction. $\square$

EXAMPLE — Convergence comparison: $\kappa = 10$ vs $\kappa = 100$

To reduce $\|\square_T - \square^*\|^2$ below $\epsilon = 10^{-4}$ starting from $\|\square_0 - \square^*\|^2 = 1$:

$$T \geq \frac{\ln(1/\epsilon)}{\ln(1/(1 - 1/\kappa))} \approx \kappa \ln(1/\epsilon)$$

$\kappa = 10$: $T \approx 10 \times \ln(10^4) \approx 92$ steps.

$\kappa = 100$: $T \approx 100 \times \ln(10^4) \approx 921$ steps.

Preconditioning (e.g. Adam, which estimates κ per coordinate) aims to reduce the effective κ and thus the number of steps required.

Non-convex case and stationary points. For non-convex f (the typical case in deep learning), gradient descent converges to a *stationary point* $\|\nabla f(\square_T)\|^2 \rightarrow 0$, but not necessarily a global minimum. The convergence rate for non-convex L -smooth f :

$$\min_{t=0, \dots, T-1} \|\nabla f(\square_t)\|^2 \leq \frac{2L(f(\square_0) - f_{\inf})}{T}$$

where $f_{\inf} = \inf_{\square} f(\square)$. In practice, deep networks have surprisingly benign non-convex landscapes: saddle points are exponentially rare in high dimensions (Dauphin et al., 2014), and most local minima have similar loss values near the global minimum.

5.5 Learning Rate Warmup and Cosine Annealing

Linear warmup. At the start of training, parameter estimates are poor and gradients are noisy. A large initial learning rate causes instability. Linear warmup increases η from 0 to $\eta_{\max}$ over T_{warm} steps:

$$\eta_t = \frac{t}{T_{\text{warm}}} \eta_{\max}, \quad t = 1, \dots, T_{\text{warm}}$$

Typical: $T_{\text{warm}} = 1000\text{--}10,000$ steps. Used in all major LLM training runs (BERT, GPT-3, LLaMA).

Cosine annealing. After warmup, the learning rate decays following a cosine curve:

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min}) \left(1 + \cos \frac{\pi(t - T_{\text{warm}})}{T - T_{\text{warm}}} \right)$$

This smoothly interpolates from $\eta_{\max}$ at $t = T_{\text{warm}}$ to $\eta_{\min}$ at $t = T$. Setting $\eta_{\min} = 0.1\eta_{\max}$ is common.

Why cosine? Near the end of training, a slow decay lets the optimiser explore a flat basin and find a flatter minimum, which generalises better (Hochreiter & Schmidhuber, 1997; Li et al., 2020).

EXAMPLE — Warmup + Cosine Schedule: Step-by-Step Values

$$\eta_{\max} = 3 \times 10^{-4}, \eta_{\min} = 3 \times 10^{-5}, T_{\text{warm}} = 2000, T = 100,000.$$

Step t	η_t
1,000	1.5×10^{-4} (warmup midpoint)
2,000	3.0×10^{-4} (warmup peak)
50,000	1.65×10^{-4} (cosine midpoint)
99,000	3.3×10^{-5} (near end)
100,000	3.0×10^{-5} (minimum)

Cyclic learning rates and SGDR. Loshchilov & Hutter (2017) introduced SGD with Warm Restarts (SGDR): the cosine schedule is periodically reset with increasing period $T_i = T_0 \cdot 2^i$. Each restart acts as a “hop” to a new basin, enabling exploration of multiple local minima. Ensembles of the checkpoints taken at each restart (called *snapshot ensembles*) often outperform a single model.

5.6 Gradient Clipping and Training Stability

Gradient norm clipping. If $\|\nabla_{\theta} \mathcal{L}\|_2 > \tau$, rescale:

$$\nabla_{\theta} \mathcal{L} \leftarrow \frac{\tau}{\|\nabla_{\theta} \mathcal{L}\|_2} \nabla_{\theta} \mathcal{L}$$

Typical threshold $\tau \in \{0.5, 1.0, 5.0\}$. This preserves direction but caps magnitude. Used universally in LLM training; GPT-2 uses $\tau = 1.0$.

Why clipping is necessary. In RNNs (§8.2), the BPTT gradient involves products of weight matrices over many time steps. When $\|\mathbf{W}\| > 1$, these products grow exponentially. A single corrupted batch (outlier loss spike) can send $\|\nabla\|$ to 10^5 or more, causing a catastrophic update $\Delta\theta \sim 10^5 \eta$ that destroys previously learned representations.

Value clipping. An alternative clips individual gradient components:

$$[\nabla_{\theta}\mathcal{L}]_i \leftarrow \text{clip}([\nabla_{\theta}\mathcal{L}]_i, -\tau, +\tau)$$

Value clipping changes the gradient direction (unlike norm clipping), which can introduce bias. Norm clipping is preferred in practice because it preserves the update direction.

Numerical example. Consider a gradient vector $\mathbf{g} = (3.0, 4.0)^\top$ with $\tau = 1.0$:

- Norm: $\|\mathbf{g}\|_2 = 5.0 > \tau = 1.0$.
- After norm clipping: $\mathbf{g}' = \frac{1.0}{5.0}(3.0, 4.0)^\top = (0.6, 0.8)^\top$. Direction preserved.
- After value clipping: $\mathbf{g}'' = (1.0, 1.0)^\top$. Direction changed (from $\arctan(4/3) \approx 53^\circ$ to 45°).

Monitoring gradient norms. In practice, the gradient norm $\|\nabla_{\theta}\mathcal{L}\|_2$ is logged at every step. A stable training run shows a norm that decreases or stays approximately constant over time. Sudden spikes (“loss spikes”) correlate with gradient norm explosions and are diagnosed by this metric. Setting τ too small (e.g. $\tau = 0.1$ when typical norm is 0.5) constantly clips and slows learning; too large ($\tau = 100$) provides no protection.

Gradient clipping does not fix the root cause (ill-conditioning or unstable dynamics). It is a safety mechanism. For RNNs, proper initialisation, gating (LSTM/GRU), or replacing recurrence with attention (Transformer) is the structural solution.

5.7 Second-Order Optimisation Methods

First-order methods (SGD, Adam) use only gradient information. Second-order methods additionally use curvature (the Hessian $\mathbf{H} = \nabla^2\mathcal{L}$), allowing larger, better-directed steps. They are rarely used in full form for deep learning (the Hessian has P^2 entries for P parameters), but approximations are important.

Newton’s method. The second-order Taylor expansion of $\mathcal{L}$ around $\square_t$:

$$\mathcal{L}(\square_t + \Delta\square) \approx \mathcal{L}(\square_t) + \nabla\mathcal{L}^\top\Delta\square + \frac{1}{2}\Delta\square^\top\mathbf{H}\Delta\square$$

Minimising over $\Delta\square$: $\mathbf{H}\Delta\square = -\nabla\mathcal{L}$, so the Newton step is:

$$\square_{t+1} = \square_t - \mathbf{H}^{-1}\nabla\mathcal{L}$$

Newton’s method has quadratic convergence near a minimum. However, computing and inverting $\mathbf{H} \in \mathbb{R}^{P \times P}$ costs $O(P^2)$ storage and $O(P^3)$ computation — infeasible for $P = 10^8$.

Quasi-Newton methods (L-BFGS). Approximate $\mathbf{H}^{-1}$ using a low-rank update from the history of gradients, without ever forming the full Hessian. L-BFGS stores the last $m \approx 10$ –20 gradient differences and uses them to implicitly represent $\mathbf{H}^{-1}$. Memory: $O(mP)$. L-BFGS is the standard optimiser for small-to-medium batch or full-batch problems (e.g. fine-tuning linear layers, optimising physics simulations), but requires large batch sizes (full-batch or very large mini-batch) to work well.

Natural gradient. The natural gradient (Amari, 1998) preconditions by the Fisher information matrix $\mathbf{F}$:

$$\tilde{\nabla}\mathcal{L} = \mathbf{F}^{-1}\nabla\mathcal{L}, \quad \mathbf{F} = \mathbb{E}[\nabla\log p(y|\mathbf{x};\theta)(\nabla\log p(y|\mathbf{x};\theta))^\top]$$

$\mathbf{F}$ is the Riemannian metric on the statistical manifold of the model’s output distributions. The natural gradient update is invariant to reparameterisation — a desirable property that standard gradient descent lacks. K-FAC (Kronecker-Factored Approximate Curvature, Martens & Grosse, 2015) approximates $\mathbf{F}^{-1}$ as a product of Kronecker factors, making the natural gradient tractable for neural networks (Volume IV).

Method	Memory	Per-step cost	Convergence	Use case
SGD	$O(P)$	$O(P)$	$O(1/T)$ or linear	Large scale
Adam	$O(P)$	$O(P)$	Empirically fast	Default choice
L-BFGS	$O(mP)$	$O(mP)$	Super-linear	Full-batch, small P
Newton	$O(P^2)$	$O(P^3)$	Quadratic	Very small P
K-FAC	$O(P)$	$O(P^{3/2})$ approx	Fast	LLM fine-tuning

Regularization

Overfitting occurs when a model memorises training data. We derive three fundamental regularization techniques: L1/L2 weight decay, Dropout, and Batch Normalization.

6.1 L1 and L2 Regularization

Regularisation prevents overfitting by penalising model complexity in the loss function. The simplest and most widely used forms are L1 and L2.

L1 and L2 Penalties

$$\mathcal{L}_{\text{L2}} = \mathcal{L}_0 + \frac{\lambda}{2} \|\theta\|_2^2 = \mathcal{L}_0 + \frac{\lambda}{2} \sum_j \theta_j^2$$

$$\mathcal{L}_{\text{L1}} = \mathcal{L}_0 + \lambda \|\theta\|_1 = \mathcal{L}_0 + \lambda \sum_j |\theta_j|$$

where $\mathcal{L}_0$ is the original loss and $\lambda > 0$ is the regularisation coefficient.

Effect of L2 on the update. The gradient of the L2 penalty is $\lambda\theta$. The update becomes:

$$\theta_{t+1} = \theta_t - \eta \nabla \mathcal{L}_0 - \eta \lambda \theta_t = (1 - \eta \lambda) \theta_t - \eta \nabla \mathcal{L}_0$$

The factor $(1 - \eta \lambda) < 1$ *decays* the weight at every step: this is why L2 is also called *weight decay*. It pushes weights toward zero but never exactly to zero.

Effect of L1 on the update. The gradient of $|\theta_j|$ is $\text{sign}(\theta_j)$ for $\theta_j \neq 0$. The update:

$$\theta_{t+1} = \theta_t - \eta \nabla \mathcal{L}_0 - \eta \lambda \text{sign}(\theta_t)$$

L1 subtracts a *constant* amount at every step, eventually driving θ_j exactly to zero. It produces *sparse* solutions: many weights are exactly zero.

EXAMPLE — L1 vs L2 Comparison

Parameter $\theta = 0.1$, $\eta = 0.01$, $\lambda = 0.1$, loss gradient $\nabla \mathcal{L}_0 = 0$ (limiting case).

L2: $\theta_{\text{new}} = 0.1 \cdot (1 - 0.01 \cdot 0.1) = 0.1 \cdot 0.999 = 0.0999$. Proportional reduction.

L1: $\theta_{\text{new}} = 0.1 - 0.01 \cdot 0.1 \cdot \text{sign}(0.1) = 0.1 - 0.001 = 0.099$. Constant reduction.

With L1, after ≈ 100 steps $\theta = 0$ (zeroed out). With L2, $\theta_t = 0.1 \cdot 0.999^t \rightarrow 0$ but asymptotically.

Bayesian interpretation.

- **L2** corresponds to a Gaussian prior $p(\theta) = \mathcal{N}(0, 1/\lambda)$: weights are normally distributed around zero.
- **L1** corresponds to a Laplace prior $p(\theta_j) \propto e^{-\lambda|\theta_j|}$: heavier tails promote sparsity.

Elastic Net. The L1+L2 combination is the *Elastic Net*:

$$\mathcal{L}_{\text{EN}} = \mathcal{L}_0 + \alpha \lambda \|\theta\|_1 + \frac{(1 - \alpha) \lambda}{2} \|\theta\|_2^2$$

It balances sparsity (L1) and stability (L2); used in linear regression and sometimes in sparse neural networks.

Weight decay vs L2 in Adam. With SGD, L2 regularisation and weight decay are equivalent. With Adam, they are not: Adam scales the gradient by $\sqrt{\hat{v}}$, and the same scaling is applied to the L2 penalty, distorting its effect. AdamW (Section 5.3) decouples the two concepts by applying weight decay directly to the weights, independently of the adaptive learning rate.

6.2 Dropout

During training, randomly set each neuron output to zero with probability p (typically $p \in [0.2, 0.5]$):

$$\tilde{a}_i^{(l)} = \frac{m_i \cdot a_i^{(l)}}{1 - p}, \quad m_i \sim \text{Bernoulli}(1 - p)$$

The division by $(1 - p)$ is *inverted dropout*: it ensures $\mathbb{E}[\tilde{a}_i] = a_i$, so no rescaling is needed at test time (at test time, use all neurons without masking).

Mathematical formulation. Let $\mathbf{m}^{(l)} \in \{0, 1\}^{n_l}$ be the dropout mask drawn independently at each training step. The masked layer output is:

$$\tilde{\mathbf{a}}^{(l)} = \frac{1}{1 - p} \mathbf{m}^{(l)} \odot \sigma(\mathbf{W}^{(l)} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)})$$

The expected value $\mathbb{E}_{\mathbf{m}}[\tilde{\mathbf{a}}^{(l)}] = \mathbf{a}^{(l)}$ (the undropped activation), which is what is computed at inference time. Variance: $\text{Var}[\tilde{a}_i] = \frac{p}{1-p} (a_i^{(l)})^2$ — dropout adds multiplicative noise proportional to activation magnitude.

Ensemble interpretation. A network with n_{hidden} neurons and dropout defines an implicit ensemble over $2^{n_{\text{hidden}}}$ sub-networks (all possible masks). Each training step trains a different sub-network drawn from this ensemble. The prediction at test time (full network with rescaled weights) approximates the geometric mean of the ensemble's predictions for classification (Srivastava et al., 2014).

Variance reduction. Consider a single output unit $y = \sum_i w_i a_i$. With dropout:

$$\text{Var} \left[\sum_i \frac{m_i w_i a_i}{1 - p} \right] = \frac{p}{1 - p} \sum_i w_i^2 a_i^2$$

This regularisation discourages co-adaptation: a neuron cannot rely on specific other neurons being present, so it must learn features useful in isolation. The gradient update for w_i is zero whenever $m_i = 0$, meaning each weight is updated only $\sim (1 - p)$ fraction of training steps.

Connection to ℓ_2 regularisation. Wager et al. (2013) showed that for generalised linear models, dropout gradient updates are equivalent to adaptive ℓ_2 regularisation with weight-specific regularisation strength proportional to the feature variance. For neural networks, the equivalence is approximate.

Practical considerations.

- Use $p = 0.5$ for fully-connected layers, $p = 0.1$ – 0.2 for convolutional layers (spatial dropout).
- Do not apply dropout to the output layer.
- For Transformers, dropout is applied to attention weights and FFN intermediate activations (typically $p = 0.1$).

- `model.train()` enables dropout; `model.eval()` disables it — forgetting to call `model.eval()` during inference is a common bug.

6.3 Batch Normalization — Every Step

Batch Normalisation (Ioffe & Szegedy, 2015) addresses *internal covariate shift*: as parameters update, the distribution of each layer's input changes, forcing subsequent layers to continuously adapt. BN normalises layer inputs to have zero mean and unit variance within each mini-batch.

Training vs. inference distinction. During training, BN uses *batch statistics*. During inference (where we may process single examples), it uses *running statistics* accumulated during training:

$$\mu_{\text{run}} \leftarrow (1 - \alpha)\mu_{\text{run}} + \alpha\mu_{\mathcal{B}}, \quad \sigma_{\text{run}}^2 \leftarrow (1 - \alpha)\sigma_{\text{run}}^2 + \alpha\sigma_{\mathcal{B}}^2$$

with momentum $\alpha = 0.1$ typically. At inference: $\hat{x}_i = (x_i - \mu_{\text{run}}) / \sqrt{\sigma_{\text{run}}^2 + \epsilon}$, then $y_i = \gamma\hat{x}_i + \beta$.

A common bug: forgetting to call `model.eval()` before inference. In eval mode, PyTorch uses $\mu_{\text{run}}, \sigma_{\text{run}}^2$. In train mode, it uses batch statistics — which are meaningless for batch size 1 (e.g. single-image inference), causing wildly incorrect outputs.

Batch Normalisation (Ioffe & Szegedy, 2015) stabilises and accelerates training by normalising intermediate activations. It is one of the most impactful contributions in the history of deep learning.

Motivating problem: Internal Covariate Shift. During training, the distribution of inputs to each layer changes as the parameters of previous layers are updated. This forces each layer to continuously adapt to shifting distributions, slowing convergence.

Batch Normalisation — Complete Formulas

For a mini-batch $\mathcal{B} = \{z_1, \dots, z_B\}$ of pre-normalisation activations (scalars for simplicity):

Forward pass:

1. Batch mean: $\mu_{\mathcal{B}} = \frac{1}{B} \sum_{i=1}^B z_i$
2. Batch variance: $\sigma_{\mathcal{B}}^2 = \frac{1}{B} \sum_{i=1}^B (z_i - \mu_{\mathcal{B}})^2$
3. Normalisation: $\hat{z}_i = \frac{z_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$
4. Re-scaling: $y_i = \gamma\hat{z}_i + \beta$

where γ, β are learned parameters; $\epsilon \approx 10^{-5}$ prevents division by zero.

BN backward pass. The derivation of the BN backward pass is one of the most involved in deep learning. The gradient with respect to z_i is:

$$\frac{\partial \mathcal{L}}{\partial z_i} = \frac{\gamma}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \left[\frac{\partial \mathcal{L}}{\partial \hat{z}_i} - \frac{1}{B} \sum_{j=1}^B \frac{\partial \mathcal{L}}{\partial \hat{z}_j} - \frac{\hat{z}_i}{B} \sum_{j=1}^B \frac{\partial \mathcal{L}}{\partial \hat{z}_j} \hat{z}_j \right]$$

The three terms correspond, respectively, to the direct contribution, the dependence through the mean, and the dependence through the variance.

EXAMPLE — BN Numerical Example on a Batch of 4

Batch: $z = (-2.0, -1.0, 0.0, 1.0)$, $\gamma = 1$, $\beta = 0$.

$$\mu = (-2 - 1 + 0 + 1)/4 = -0.5$$

$$\sigma^2 = [(-2+0.5)^2 + (-1+0.5)^2 + (0+0.5)^2 + (1+0.5)^2]/4 = [2.25 + 0.25 + 0.25 + 2.25]/4 = 1.25$$

$$\hat{z} = (-2 - (-0.5))/\sqrt{1.25 + \epsilon} \approx -1.5/1.118 = -1.342$$

$$\hat{z}_2 \approx -0.447, \hat{z}_3 \approx 0.447, \hat{z}_4 \approx 1.342$$

Result: mean ≈ 0 , variance ≈ 1 — regardless of the original values.

Learned parameters γ and β . If BN only normalised, it would remove the expressive capacity of the layer (e.g., a sigmoid would require unit-variance inputs to operate in its linear region). γ and β allow the network to restore any distribution optimal for training.

Training vs Inference. At inference time, the batch may be of size 1 (e.g., single-sample prediction). The *running mean* and *running variance* statistics accumulated during training are used:

$$\mu_{\text{run}} \leftarrow (1 - \alpha)\mu_{\text{run}} + \alpha\mu_{\mathcal{B}}, \quad \alpha=0.1$$

BN and batch size. BN works well for $B \geq 32$. For small batches, the statistics are noisy and BN degrades. Alternatives: Layer Normalisation (normalises over features, not the batch — standard in Transformers), Group Normalisation (divides channels into groups), Instance Normalisation (used in style transfer).

6.4 Layer Normalisation vs. Batch Normalisation

$$\text{LN: } \hat{z}_j = \frac{z_j - \mu}{\sqrt{\sigma^2 + \epsilon}}, \quad \mu = \frac{1}{n} \sum_j z_j, \quad \sigma^2 = \frac{1}{n} \sum_j (z_j - \mu)^2$$

Property	BatchNorm	LayerNorm
Normalisation axis	Batch	Feature
Works with $B = 1$	No	Yes
Used in	CNNs, MLPs	Transformers, RNNs
Test-time behaviour	Running stats	Computed per example

RMSNorm (LLaMA, GPT-NeoX): $\hat{z}_j = z_j/\text{RMS}(\mathbf{z}) \cdot \gamma_j$ where $\text{RMS}(\mathbf{z}) = \sqrt{\frac{1}{n} \sum_j z_j^2}$. No mean subtraction — computationally cheaper.

6.5 Stochastic Depth

Proposed by Huang et al. (2016), stochastic depth randomly drops entire residual blocks during training. Let block $l \in \{1, \dots, L\}$ have survival probability:

$$p_l = 1 - \frac{l}{L}(1 - p_{\text{final}})$$

Linear schedule: early layers (small l) survive with probability near 1; the last layer survives with probability p_{final} (typically 0.5). This reflects the intuition that deeper layers learn more specialised features that are more safely dropped.

Forward pass. Let $b_l \sim \text{Bernoulli}(p_l)$ be a random survival indicator. The residual block becomes:

$$\mathbf{x}_{l+1} = \mathbf{x}_l + b_l \cdot \mathcal{F}_l(\mathbf{x}_l)$$

At inference: replace b_l with its expectation p_l :

$$\mathbf{x}_{l+1} = \mathbf{x}_l + p_l \cdot \mathcal{F}_l(\mathbf{x}_l)$$

Effects.

- *Regularisation:* the network must learn robust representations that do not depend on any single block.
- *Implicit ensemble:* the 2^L possible subnetworks (each with different dropped blocks) form an ensemble. The final model averages over all of them via the p_l scaling.
- *Training speedup:* dropped blocks are skipped in both forward and backward passes, reducing effective training FLOPs by $\sim (1 - p_{\text{final}})/2$.
- *Gradient flow:* when block l is dropped, gradient flows directly through the skip connection from $\mathbf{x}_{l+1}$ to $\mathbf{x}_l$ without passing through $\mathcal{F}_l$ — further improving deep gradient flow.

Used in DeiT and modern vision transformers with $p_{\text{final}} = 0.1$ to 0.5 depending on model depth.

6.6 Why Dropout Works: Ensemble Interpretation

Dropout with rate p trains 2^N different sub-networks (one per binary mask on N neurons) simultaneously, sharing parameters. At test time, the full network with weights scaled by $1 - p$ approximates the geometric mean of predictions from all sub-networks.

Formal bound. Let $\hat{y}_{\text{mask}}(\mathbf{x})$ be the output with mask $\mathbf{m} \in \{0, 1\}^N$, $m_i \sim \text{Bernoulli}(1 - p)$ i.i.d. The test-time output (no dropout) approximates:

$$\hat{y}_{\text{test}} \approx \mathbb{E}_{\mathbf{m}}[\hat{y}_{\text{mask}}(\mathbf{x})]$$

This expectation over exponentially many masks is approximated by the single forward pass with scaled weights.

Regularisation effect. Co-adaptation: without dropout, neuron i may learn to correct errors of neuron j . Dropping either forces each neuron to be independently useful. This is analogous to injecting noise $\tilde{\mathbf{h}} = \mathbf{h} \odot \mathbf{m} / (1 - p)$ into activations, which acts as a Tikhonov regulariser on the weight norms (Wager et al., 2013).

EXAMPLE — Dropout at Training and Test Time

Hidden layer $\mathbf{h} = (2.0, 0.5, -1.5, 3.0)$, $p = 0.5$ (drop probability).

Training (one sample mask): mask = $(1, 0, 1, 0)$, scaled output = $(2.0/0.5, 0, -1.5/0.5, 0) = (4.0, 0, -3.0, 0)$.

Test time: no mask, weights already absorbed: $\mathbf{h}_{\text{test}} = \mathbf{h} = (2.0, 0.5, -1.5, 3.0)$.

The inverted dropout (divide by $1 - p$ during training) ensures the expected activation magnitude is the same at train and test time, avoiding the need for weight rescaling at inference.

6.7 Batch Normalisation: Full Derivation of Backpropagation

Let $\mathbf{x} \in \mathbb{R}^B$ be the pre-activations for a single feature over a batch. The BN forward pass:

$$\mu_{\mathcal{B}} = \frac{1}{B} \sum_{i=1}^B x_i, \quad \sigma_{\mathcal{B}}^2 = \frac{1}{B} \sum_{i=1}^B (x_i - \mu_{\mathcal{B}})^2, \quad \hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}, \quad y_i = \gamma \hat{x}_i + \beta$$

Backward pass. Let $\bar{y}_i = \partial \mathcal{L} / \partial y_i$ (upstream). We need $\partial \mathcal{L} / \partial x_i$.

Step 1: $\partial \mathcal{L} / \partial \hat{x}_i = \bar{y}_i \cdot \gamma$.

Step 2: $\partial \mathcal{L} / \partial \sigma_{\mathcal{B}}^2 = \sum_i \bar{y}_i \gamma \cdot (x_i - \mu_{\mathcal{B}}) \cdot (-\frac{1}{2}) (\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$.

Step 3: $\partial \mathcal{L} / \partial \mu_{\mathcal{B}} = \sum_i \bar{y}_i \gamma \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \cdot \frac{\partial \mathcal{L}}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{-2}{B} \sum_i (x_i - \mu_{\mathcal{B}})$.

Step 4 (combining):

$$\bar{x}_i = \frac{\bar{y}_i \gamma}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\frac{\partial \mathcal{L}}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{B} + \frac{\partial \mathcal{L}}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{B}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

Note that each $\bar{x}_i$ depends on *all* batch elements through $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$. This batch-coupling makes BN computationally expensive and causes problems with very small batch sizes (statistics noisy) or distributed training (cross-device batch statistics require communication).

6.8 Data Augmentation as Regularisation

Data augmentation reduces overfitting by artificially expanding the training set with label-preserving transformations. It is one of the most effective regularisers in computer vision and, increasingly, in NLP.

Formal viewpoint. Let $\mathcal{T}$ be a distribution over transformations $T : \mathcal{X} \rightarrow \mathcal{X}$ such that $f^*(\mathbf{x}) = f^*(T(\mathbf{x}))$ for the true labelling function f^* . Data augmentation trains on the augmented distribution:

$$\tilde{P}(\mathbf{x}, y) = \mathbb{E}_{T \sim \mathcal{T}} [P(T^{-1}(\mathbf{x}), y)]$$

The loss minimised is $\mathbb{E}_{T \sim \mathcal{T}} [\mathcal{L}(f(T(\mathbf{x})), y)]$ — the expected loss over random transforms. This is equivalent to training on an infinitely large dataset of augmented examples (sampling a new transform each epoch).

Standard augmentations for image classification.

Augmentation	Operation	Effect
Random crop	Sample region of size $s \times s$ from $H \times W$ image	Translation invariance
Horizontal flip	Mirror image with prob. 0.5	Reflection symmetry
Colour jitter	Perturb brightness, contrast, saturation	Lighting invariance
Random erasing	Set rectangle to random noise	Occlusion robustness
Gaussian blur	Convolve with $\mathcal{N}(0, \sigma^2)$ kernel	Frequency regularisation
Mixup	$\tilde{x} = \lambda x_i + (1 - \lambda) x_j, \tilde{y} = \lambda y_i + (1 - \lambda) y_j$	Label smoothing

Mixup. The Mixup augmentation (Zhang et al., 2018) interpolates between pairs of training examples and their labels:

$$\tilde{\mathbf{x}} = \lambda \mathbf{x}_i + (1 - \lambda) \mathbf{x}_j, \quad \tilde{\mathbf{y}} = \lambda \mathbf{y}_i + (1 - \lambda) \mathbf{y}_j, \quad \lambda \sim \text{Beta}(\alpha, \alpha)$$

Training on mixed examples forces the network to behave linearly between training points, penalising sharp decision boundaries. Mixup with $\alpha = 0.2$ reduces test error on CIFAR-10 by $\approx 0.5\%$ and on ImageNet by $\approx 0.3\%$, and substantially improves calibration.

CutMix. Cuts a rectangular patch from one image and pastes it onto another; the label is mixed proportionally to the area ratio. Combined with Mixup and other augmentations in the AugMix/RandAugment pipeline, data augmentation accounts for 1–3% absolute accuracy improvement on ImageNet.

Augmentation in NLP. Equivalent augmentations exist for text:

- *Back-translation*: translate to another language and back.
- *Token dropout*: randomly delete tokens (analogous to erasing).
- *Synonym substitution*: replace words with synonyms (EDA, Wei & Zou 2019).
- *Span masking* (used in T5, BART): mask contiguous spans of tokens and train to reconstruct.

Data augmentation and dropout both add noise during training, but at different levels: dropout adds noise at the representation level (zeroing neurons), while augmentation adds noise at the input level (transforming data). Both discourage memorisation of specific training examples, but through different mechanisms. The two are complementary and are used together in modern architectures.

Convolutional Neural Networks

We define the discrete convolution rigorously, derive the convolution backward pass, explain pooling and its gradient, and analyse residual connections.

7.1 The Discrete Convolution — Every Step

The discrete convolution is the fundamental operation of CNNs. It exploits two invariance principles for visual pattern recognition: *stationarity* (patterns appear anywhere in the image) and *locality* (relevant features are local).

1D Convolution

For a signal $x \in \mathbb{R}^n$ and a kernel $w \in \mathbb{R}^k$, the convolution (actually cross-correlation as implemented in ML) is:

$$(x \star w)_j = \sum_{m=0}^{k-1} w_m \cdot x_{j+m}, \quad j = 0, \dots, n-k$$

Output size: $n - k + 1$ (without padding).

2D Convolution for Images

For an image $\mathbf{X} \in \mathbb{R}^{H \times W}$ and a kernel $\mathbf{W} \in \mathbb{R}^{k_H \times k_W}$:

$$(\mathbf{X} \star \mathbf{W})_{i,j} = \sum_{p=0}^{k_H-1} \sum_{q=0}^{k_W-1} W_{p,q} \cdot X_{i+p,j+q}$$

Output shape: $(H - k_H + 1) \times (W - k_W + 1)$.

With C_{in} input channels and C_{out} filters, the conv layer has:

$$C_{\text{out}} \times C_{\text{in}} \times k_H \times k_W + C_{\text{out}} \text{ parameters}$$

$\boxed{\{Z\}}$
weights

$|\{Z\}|$
biases

EXAMPLE — 3×3 Convolution on a 5×5 Image

Image $\mathbf{X}$ (one channel, values 0–8):

$$\mathbf{X} = \begin{pmatrix} 1 & 2 & 3 & 0 & 1 \\ 2 & 3 & 4 & 1 & 2 \\ 3 & 4 & 5 & 2 & 3 \\ 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}, \quad \mathbf{W} = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

The kernel $\mathbf{W}$ is the horizontal Sobel filter (detects vertical edges).

Output (0,0): $1 \cdot 1 + 0 \cdot 2 + (-1) \cdot 3 + 2 \cdot 2 + 0 \cdot 3 + (-2) \cdot 4 + 1 \cdot 3 + 0 \cdot 4 + (-1) \cdot 5 = 1 - 3 + 4 - 8 + 3 - 5 = -8$

Output (0,1): $1 \cdot 2 + 0 \cdot 3 + (-1) \cdot 0 + 2 \cdot 3 + 0 \cdot 4 + (-2) \cdot 1 + 1 \cdot 4 + 0 \cdot 5 + (-1) \cdot 2 = 2 - 0 + 6 - 2 + 4 - 2 = 8$

The 3×3 output highlights horizontal intensity variations.

Parameter sharing and translation equivariance. Every output neuron uses the same kernel $\mathbf{W}$: this is *parameter sharing*, which drastically reduces parameters compared to a fully-connected layer. The consequence is *translation equivariance*: if the input shifts by (p, q) , the output shifts by the same vector. Formally: $\mathcal{F}[T_{(p,q)}\mathbf{X}] = T_{(p,q)}[\mathcal{F}[\mathbf{X}]]$.

Padding and Stride.

- **Padding** p : adds p rows/columns of zeros to the borders. “Same” padding ($p = (k - 1)/2$) preserves the spatial dimension.
- **Stride** s : moves the kernel s pixels at a time. Reduces the output dimension by a factor s .
- Output size (1D): $\lfloor (n + 2p - k)/s \rfloor + 1$.

Backpropagation through convolution. Let $\mathbf{Y} = \mathbf{X} \star \mathbf{W}$ (valid convolution, no padding). Given upstream gradient $\bar{\mathbf{Y}} = \partial\mathcal{L}/\partial\mathbf{Y}$:

Gradient w.r.t. kernel:

$$\bar{W}_{p,q} = \sum_{i,j} \bar{Y}_{i,j} \cdot X_{i+p,j+q}$$

This is a convolution of $\mathbf{X}$ with $\bar{\mathbf{Y}}$.

Gradient w.r.t. input:

$$\bar{X}_{i,j} = \sum_{p,q} \bar{Y}_{i-p,j-q} \cdot W_{p,q}$$

This is a full convolution of $\bar{\mathbf{Y}}$ with the *flipped* kernel $\mathbf{W}_{p,q} \rightarrow \mathbf{W}_{k-1-p,k-1-q}$ — equivalently, a convolution with the transposed (180°-rotated) kernel.

EXAMPLE — Gradient of Convolution, 3×3 kernel on 4×4 input

Input $\mathbf{X} \in \mathbb{R}^{4 \times 4}$, kernel $\mathbf{W} \in \mathbb{R}^{3 \times 3}$, output $\mathbf{Y} \in \mathbb{R}^{2 \times 2}$.

Suppose $\bar{\mathbf{Y}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ (identity upstream gradient for simplicity).

$$\bar{W}_{0,0} = \bar{Y}_{0,0} \cdot X_{0,0} + \bar{Y}_{0,1} \cdot X_{0,1} + \bar{Y}_{1,0} \cdot X_{1,0} + \bar{Y}_{1,1} \cdot X_{1,1} = X_{0,0} + X_{1,1}.$$

This shows that the kernel gradient accumulates contributions from all spatial positions of the input that were multiplied by $W_{0,0}$ during the forward pass — parameter sharing means the kernel gradient is a sum of $|\mathbf{Y}|$ outer products.

Convolution gradient. The gradient with respect to the weights is a convolution of the input signal with the output gradient; the gradient with respect to the input is a convolution of the output gradient with the *flipped* kernel (true convolution, not cross-correlation).

7.2 Pooling and Its Backpropagation

Max Pooling with window $k \times k$ and stride S :

$$Y_{i,j} = \max_{0 \leq p,q < k} X_{i \cdot S + p, j \cdot S + q}$$

Backward pass. The gradient flows only to the maximum element in each pool window. If $X_{r,s}$ was the maximum in its window:

$$\frac{\partial \mathcal{L}}{\partial X_{r,s}} = \frac{\partial \mathcal{L}}{\partial Y_{i,j}}, \quad \frac{\partial \mathcal{L}}{\partial X_{r',s'}} = 0 \quad \text{for non-max positions}$$

Average Pooling: $Y_{i,j} = \frac{1}{k^2} \sum_{p,q} X_{i \cdot S + p, j \cdot S + q}$. Backward: gradient distributes uniformly:
 $\frac{\partial \mathcal{L}}{\partial X_{r,s}} = \frac{1}{k^2} \frac{\partial \mathcal{L}}{\partial Y_{i,j}}$.

Global Average Pooling (GAP). A special case of average pooling where the window covers the entire spatial extent: for input $\mathbf{X} \in \mathbb{R}^{C \times H \times W}$,

$$\text{GAP}(\mathbf{X})_c = \frac{1}{HW} \sum_{i=1}^H \sum_{j=1}^W X_{c,i,j}$$

Output: a vector of C scalars (one per channel). GAP collapses the spatial dimensions entirely, producing a compact representation. It is used in modern architectures (ResNet, MobileNet, EfficientNet) instead of fully connected layers, reducing parameters from $C \times H \times W \times d_{\text{out}}$ to $C \times d_{\text{out}}$ and improving spatial invariance.

Why max pooling is non-differentiable. At the pool boundary — when two inputs tie for the maximum — the gradient is not uniquely defined. In practice, a sub-gradient is used: gradient goes to one arbitrarily chosen maximum position (e.g., the first). This does not cause problems in training because ties have measure zero under continuous distributions.

Output spatial dimensions. For a $H_{\text{in}} \times W_{\text{in}}$ feature map with pooling window $k \times k$, stride S , and padding P :

$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} + 2P - k}{S} \right\rfloor + 1$$

Standard max pooling: $k = 2, S = 2, P = 0$ halves the spatial size ($H_{\text{out}} = H_{\text{in}}/2$) and keeps channels constant.

Adaptive pooling. `nn.AdaptiveAvgPool2d(output_size)` computes the stride and kernel size automatically to produce a given output spatial size, regardless of input size. For input H_{in} and target H_{out} : effective stride = $\lfloor H_{\text{in}}/H_{\text{out}} \rfloor$, kernel = $H_{\text{in}} - (H_{\text{out}} - 1) \cdot \text{stride}$. This enables networks to accept variable-size inputs, used in detection and segmentation models.

7.3 Residual Connections — Why They Solve the Degradation Problem

Definition: Residual Block

$$\mathbf{a}^{\text{out}} = \phi(\mathbf{W}_2 \phi(\mathbf{W}_1 \mathbf{a}^{\text{in}}) + \mathbf{a}^{\text{in}})$$

The term $\mathbf{a}^{\text{in}}$ is the *skip connection* (identity shortcut).

Degradation problem. In very deep networks without skip connections, training accuracy degrades (not just test accuracy) — the network is harder to optimise. A residual block parameterises the *residual* $\mathbf{F}(\mathbf{x}) = \mathbf{a}^{\text{out}} - \mathbf{x}$. If the optimal function is close to identity, it is easier to push $\mathbf{F} \rightarrow \mathbf{0}$ than to push the entire mapping to identity.

Gradient flow. Differentiating through a skip connection:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{\text{in}}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{\text{out}}} \cdot \left(\mathbf{I} + \frac{\partial \mathbf{F}}{\partial \mathbf{a}^{\text{in}}} \right)$$

The identity term $\mathbf{I}$ ensures that even if $\frac{\partial \mathbf{F}}{\partial \mathbf{a}^{\text{in}}} \approx 0$ (vanishing gradient through the block), the gradient still flows backward with magnitude $\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{\text{out}}}$. Residual connections effectively provide a gradient highway through hundreds of layers.

7.4 Transposed Convolution

Transposed convolutions (also called *deconvolutions*, though this terminology is imprecise) upsample feature maps by learning to spread each input value over a $K \times K$ neighbourhood. They are essential in generators (GANs) and decoder paths (U-Net, segmentation networks).

Matrix formulation. For 1D convolution with stride $s = 1$, kernel size K , the forward pass is $\mathbf{y} = \mathbf{C}\mathbf{x}$ where $\mathbf{C}$ is a $((n - K + 1) \times n)$ Toeplitz matrix. The transposed convolution computes $\mathbf{C}^\top \mathbf{x}$, which has output size n (the original input size) — it upsamples from $n - K + 1$ back to n .

Strided transposed convolution (output size). For input of spatial size H_{in} , kernel size K , stride s , and padding p :

$$H_{\text{out}} = s(H_{\text{in}} - 1) + K - 2p$$

For $H_{\text{in}} = 4$, $K = 3$, $s = 2$, $p = 0$: $H_{\text{out}} = 2 \times 3 + 3 = 9$.

Explicit construction. For stride $s = 2$: insert $s - 1 = 1$ zero between each input element, then convolve with the flipped kernel. Input $[a, b, c]$ becomes $[a, 0, b, 0, c]$; convolving with kernel $[k_1, k_2, k_3]$ gives a 7-element output. This is why transposed convolution is sometimes described as “convolution with fractional stride $1/s$.”

EXAMPLE — Transposed convolution: 1D example

Input: $[2, 1, 3]$, kernel: $[1, 2, 1]$, stride $s = 2$, no padding.

Step 1 — zero-insert: $[2, 0, 1, 0, 3]$.

Step 2 — convolve with kernel: outputs at positions 0–6:

$$[2, 4, 1 + 2, 2, 3 + 1, 6, 3] = [2, 4, 3, 2, 4, 6, 3]$$

Output size: $2(3 - 1) + 3 = 7$. ✓

Backward pass connection. During backpropagation through a stride- s convolution, the gradient with respect to the input is exactly a transposed convolution applied to the upstream gradient. This is why the operation exists: it is the natural inverse operation for gradient flow.

Parameter sharing. Like regular convolutions, transposed convolutions share weights spatially. For a $C_{\text{in}} \rightarrow C_{\text{out}}$ transposed convolution with kernel $K \times K$: parameters = $C_{\text{in}} \times C_{\text{out}} \times K^2$.

Checkerboard artifacts. Stride- $s > 1$ transposed convolution causes uneven overlap: some output positions receive contributions from $\lceil K/s \rceil^2$ input positions, others from $\lfloor K/s \rfloor^2$. This produces regular grid-like intensity patterns in generated images. Fix: replace with bilinear or nearest-neighbour upsampling (no parameters) followed by a regular 3×3 or 1×1 convolution. This decouples upsampling (resolution change) from feature transformation.

7.5 Depthwise Separable Convolutions

A standard $K \times K$ convolution mapping C_{in} input channels to C_{out} output channels mixes spatial and channel information simultaneously, at cost $O(HWK^2C_{\text{in}}C_{\text{out}})$. Depthwise separable convolution factorises this into two cheaper stages:

1. **Depthwise (spatial) convolution:** apply a separate $K \times K$ filter to each input channel independently. Output: $H' \times W' \times C_{\text{in}}$. Cost: $2H'W'K^2C_{\text{in}}$.
2. **Pointwise (1×1) convolution:** mix channels with a 1×1 convolution. Output: $H' \times W' \times C_{\text{out}}$. Cost: $2H'W'C_{\text{in}}C_{\text{out}}$.

Reduction ratio.

$$\frac{\text{FLOPs}_{\text{DWSep}}}{\text{FLOPs}_{\text{std}}} = \frac{2H'W'C_{\text{in}}(K^2 + C_{\text{out}})}{2H'W'K^2C_{\text{in}}C_{\text{out}}} = \frac{1}{C_{\text{out}}} + \frac{1}{K^2}$$

For $K = 3$, $C_{\text{out}} = 256$: ratio = $1/256 + 1/9 \approx 0.115$ — about $8.7\times$ fewer FLOPs. Used in MobileNet, Xception, EfficientNet.

EXAMPLE — MobileNetV1 FLOPs savings on 224×224 input, $C_{\text{in}} = 32$, $C_{\text{out}} = 64$

Standard conv ($K = 3$): $2 \times 112 \times 112 \times 64 \times 32 \times 9 = 462\text{M}$ FLOPs.
 Depthwise: $2 \times 112 \times 112 \times 32 \times 9 = 7.2\text{M}$. Pointwise: $2 \times 112 \times 112 \times 32 \times 64 = 51.4\text{M}$.
 Total DW-Sep: 58.6M . Ratio: $58.6/462 \approx 12.7\%$ — $7.9\times$ savings.
 MobileNetV1 accuracy on ImageNet: 70.6% top-1 vs. VGG-16: 71.5%, at $\approx 27\times$ fewer FLOPs.

Xception: depthwise separable as extreme Inception. Chollet (2017) shows that the standard Inception module can be seen as an intermediate point between a standard convolution (one channel group) and depthwise separable convolution (C_{in} channel groups). Xception replaces all Inception modules with depthwise separable convolutions preceded by pointwise mixing, achieving better accuracy than Inception-V3 on ImageNet.

Gradient flow. The depthwise stage has no cross-channel mixing, so gradients do not flow between channels through this layer. The pointwise stage restores cross-channel gradient flow. This means the depthwise layer's gradients are local to each channel — beneficial for gradient isolation but requiring careful initialisation to ensure all channels receive useful training signal.

Relationship to grouped convolutions. A depthwise convolution is a grouped convolution with group size $g = C_{\text{in}}$ (one group per input channel). For $g = 1$: standard convolution. For $g = C_{\text{in}}$: depthwise convolution. For intermediate g : the ResNeXt architecture uses $g \in \{8, 32\}$ as a compute-efficient alternative.

7.6 Backpropagation Through a Residual Block

A residual block computes $\mathbf{y} = \mathbf{x} + F(\mathbf{x})$ where F is a sequence of layers. The gradient through the block is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \bar{\mathbf{y}}^\top \left(\mathbf{I} + \frac{\partial F(\mathbf{x})}{\partial \mathbf{x}} \right)$$

The identity term $\mathbf{I}$ is the key: even if $\partial F / \partial \mathbf{x} \approx \mathbf{0}$ (vanishing gradient in the residual branch), the gradient $\bar{\mathbf{y}}$ flows back unchanged through the skip connection. This is why ResNets with 100+ layers can be trained effectively.

Gradient norm analysis. In a plain network with L layers:

$$\left\| \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(0)}} \right\| \approx \prod_{l=1}^L \|\mathbf{w}^{(l)}\|_F \cdot \|\text{diag}(\phi')\|$$

With residual connections, each factor becomes $\|(\mathbf{I} + \mathbf{J}_F)\|$; even with $\mathbf{J}_F \approx 0$ the product remains $\approx 1^L = 1$, not $\rightarrow 0$.

EXAMPLE — Gradient Flow in a 4-Block ResNet

Suppose each block's Jacobian $\mathbf{J}_F$ has spectral radius 0.1 (small, nearly vanishing). Plain network gradient after 4 blocks: $\approx 0.1^4 = 10^{-4}$. With ResNets: each factor $\|\mathbf{I} + \mathbf{J}_F\| \approx 1 + 0.1 = 1.1$ (approximately, by triangle inequality). Product: $1.1^4 \approx 1.46$. The gradient is actually amplified rather than vanished.

7.7 FLOPs Analysis: Standard vs. Depthwise Separable Convolution

Let the input feature map be $H \times W \times C_{\text{in}}$, output $H' \times W' \times C_{\text{out}}$, kernel $k \times k$.

Standard convolution FLOPs:

$$\text{FLOPs}_{\text{std}} = 2 \cdot H' \cdot W' \cdot C_{\text{out}} \cdot C_{\text{in}} \cdot k^2$$

(the factor 2 accounts for multiply and add)

Depthwise separable convolution FLOPs:

$$\text{Depthwise: } 2 \cdot H' \cdot W' \cdot C_{\text{in}} \cdot k^2$$

$$\text{Pointwise: } 2 \cdot H' \cdot W' \cdot C_{\text{in}} \cdot C_{\text{out}}$$

$$\text{Total: } 2H'W'C_{\text{in}}(k^2 + C_{\text{out}})$$

Reduction ratio:

$$\frac{\text{FLOPs}_{\text{DW}}}{\text{FLOPs}_{\text{std}}} = \frac{k^2 + C_{\text{out}}}{k^2 \cdot C_{\text{out}}} = \frac{1}{C_{\text{out}}} + \frac{1}{k^2}$$

For $k = 3$, $C_{\text{out}} = 256$: ratio = $1/256 + 1/9 \approx 0.115$ — about $8.7\times$ fewer FLOPs. MobileNetV1 achieves this with only $\approx 1\%$ accuracy drop on ImageNet.

Recurrent Neural Networks and LSTM

We derive the RNN forward pass, show analytically why gradients vanish or explode, and build LSTM gate by gate — deriving every equation from scratch.

8.1 RNN Base

A Recurrent Neural Network (RNN) processes sequences $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T$ by maintaining a hidden state $\mathbf{h}_t$ that summarises past context. Unlike feedforward networks, the same weight matrices are applied at every time step (weight sharing across time), enabling the network to generalise to sequences of arbitrary length.

State equations.

$$\begin{aligned}\mathbf{h}_t &= \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h) \\ \hat{\mathbf{y}}_t &= \text{softmax}(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y)\end{aligned}$$

where $\mathbf{W}_{hh} \in \mathbb{R}^{H \times H}$ (recurrent weights), $\mathbf{W}_{xh} \in \mathbb{R}^{H \times d}$ (input weights), $\mathbf{W}_{hy} \in \mathbb{R}^{K \times H}$ (output weights). The initial state $\mathbf{h}_0$ is typically set to $\mathbf{0}$ or learned.

Parameter count. For hidden size H , input size d , output classes K :

$$P = H^2 + Hd + H + KH + K = H(H + d + 1) + K(H + 1)$$

Notably, P does not grow with sequence length T — the same parameters are used at every step.

Unrolling. An RNN can be viewed as a very deep feedforward network with T layers and shared weights. Backpropagation through this unrolled network is called **Backpropagation Through Time** (BPTT).

Total Loss for Sequence Modelling

For a language model predicting next tokens, the total loss over sequence $\mathbf{x}_1, \dots, \mathbf{x}_T$:

$$\mathcal{L} = \sum_{t=1}^T \mathcal{L}_t(\hat{\mathbf{y}}_t, \mathbf{y}_t)$$

where $\mathbf{y}_t$ is the one-hot encoding of the $(t + 1)$ -th token. Perplexity $= \exp(\mathcal{L}/T)$ is the standard metric: lower is better, with $\exp(0) = 1$ (perfect) and $\exp(\log K) = K$ (uniform random guessing over K classes).

A recurrent neural network processes sequences $\mathbf{x}_1, \dots, \mathbf{x}_T$ by maintaining a hidden state $\mathbf{h}_t$:

$$\mathbf{h}_t = \phi(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h), \quad \mathbf{o}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y$$

with $\mathbf{h}_0 = \mathbf{0}$. Same parameters $(\mathbf{W}_{hh}, \mathbf{W}_{xh}, \mathbf{W}_{hy})$ at every time step — this is *parameter sharing*.

8.2 BPTT and Vanishing/Exploding Gradients — Derivation

Backpropagation through time (BPTT) unrolls the RNN over T steps and applies the chain rule. The gradient of the loss with respect to $\mathbf{h}_k$ (for $k < T$):

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_k} = \left(\prod_{t=k+1}^T \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \right) \frac{\partial \mathcal{L}}{\partial \mathbf{h}_T}$$

Each Jacobian factor is:

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = \mathbf{W}_{hh}^\top \text{diag}(\phi'(\mathbf{z}_t))$$

The product of $T - k$ such matrices can have eigenvalues that:

- grow exponentially (if spectral radius $\rho(\mathbf{W}_{hh}) > 1$): **exploding gradients**
- shrink exponentially (if $\rho(\mathbf{W}_{hh}) < 1$, or $|\phi'| < 1$): **vanishing gradients**

Solution to exploding: gradient clipping, $\mathbf{g} \leftarrow \mathbf{g} \cdot \min\left(1, \frac{c}{\|\mathbf{g}\|}\right)$.

Solution to vanishing: LSTM (below).

8.3 LSTM — Every Gate Explained

Long Short-Term Memory networks (Hochreiter & Schmidhuber, 1997) solve the vanishing gradient problem in standard RNNs by introducing an explicit memory mechanism controlled by gates.

LSTM — Complete Equations

At each time step t , with input $\mathbf{x}_t \in \mathbb{R}^d$ and previous hidden state $\mathbf{h}_{t-1} \in \mathbb{R}^m$:

Forget gate:

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_f) \in (0, 1)^m$$

Input gate:

$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_i) \in (0, 1)^m$$

Cell candidate:

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_c) \in (-1, 1)^m$$

Cell update:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$

Output gate:

$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_o) \in (0, 1)^m$$

Hidden state:

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

The notation $[\mathbf{h}_{t-1}; \mathbf{x}_t]$ denotes concatenation.

Interpretation of each gate.

Forget gate $\mathbf{f}_t$: Decides which part of the memory $\mathbf{c}_{t-1}$ to *forget*. If $f_{t,j} \approx 0$: channel j of the cell is zeroed out. If $f_{t,j} \approx 1$: the memory is preserved. Example: in a language model, upon encountering a full stop, the forget gate zeros out the current sentence context.

Input gate $\mathbf{i}_t$ and candidate $\tilde{\mathbf{c}}_t$: Together they decide what new information to write into the cell. $\tilde{\mathbf{c}}_t$ is the proposed content; $\mathbf{i}_t$ controls how much of it to accept.

Cell state $\mathbf{c}_t$: The LSTM's “long-term memory”. The path $\mathbf{c}_{t-1} \rightarrow \mathbf{c}_t$ passes only through

element-wise multiplications and additions — no “saturating” nonlinearity applied recursively. This is the “constant error carousel” that resolves the vanishing gradient.

Output gate $\mathbf{o}_t$: Controls which part of the cell to expose as the hidden state $\mathbf{h}_t$.

EXAMPLE — LSTM — One Numerical Step

$m = 2$, $\mathbf{h}_{t-1} = (0.5, -0.3)$, $\mathbf{x}_t = (1.0, 0.2)$, $\mathbf{c}_{t-1} = (0.8, -0.5)$.

Suppose (for simplicity) pre-computed gates: $\mathbf{f} = (0.7, 0.9)$, $\mathbf{i} = (0.4, 0.5)$, $\tilde{\mathbf{c}} = (0.6, 0.3)$, $\mathbf{o} = (0.8, 0.6)$.

$$\mathbf{c}_t = (0.7 \cdot 0.8 + 0.4 \cdot 0.6, 0.9 \cdot (-0.5) + 0.5 \cdot 0.3) = (0.56 + 0.24, -0.45 + 0.15) = (0.80, -0.30)$$

$$\mathbf{h}_t = (0.8 \cdot \tanh(0.80), 0.6 \cdot \tanh(-0.30)) = (0.8 \cdot 0.664, 0.6 \cdot (-0.291)) = (0.531, -0.175)$$

LSTM parameters. With d inputs and m hidden units, there are 4 weight matrices of size $m \times (m + d)$ and 4 bias vectors of size m : total $4m(m + d) + 4m = 4m(m + d + 1)$ parameters. For $m = 512$, $d = 256$: $4 \cdot 512 \cdot 769 = 1,575,936$ parameters per layer.

LSTM vs Transformer. LSTMs dominated NLP until 2017. Transformers outperform them on long-range dependency tasks because attention accesses all context in $O(1)$ steps, whereas LSTMs propagate information through T RNN steps with potential degradation.

8.4 Gated Recurrent Unit (GRU)

GRU Equations

$$\begin{aligned} \mathbf{r}_t &= \sigma(\mathbf{W}_r \mathbf{x}_t + \mathbf{U}_r \mathbf{h}_{t-1}) \quad (\text{reset gate}) \\ \mathbf{u}_t &= \sigma(\mathbf{W}_u \mathbf{x}_t + \mathbf{U}_u \mathbf{h}_{t-1}) \quad (\text{update gate}) \\ \tilde{\mathbf{h}}_t &= \tanh(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h (\mathbf{r}_t \odot \mathbf{h}_{t-1})) \quad (\text{candidate}) \\ \mathbf{h}_t &= (1 - \mathbf{u}_t) \odot \mathbf{h}_{t-1} + \mathbf{u}_t \odot \tilde{\mathbf{h}}_t \quad (\text{output}) \end{aligned}$$

Intuition. The GRU (Cho et al., 2014) simplifies LSTM by merging the cell state and hidden state into a single hidden state $\mathbf{h}_t \in \mathbb{R}^d$.

- **Update gate $\mathbf{u}_t \in (0, 1)^d$:** controls the interpolation between old state $\mathbf{h}_{t-1}$ and new candidate $\tilde{\mathbf{h}}_t$. When $u_{t,i} \approx 0$: unit i carries forward its old value (memory). When $u_{t,i} \approx 1$: unit i adopts the new candidate (update).
- **Reset gate $\mathbf{r}_t \in (0, 1)^d$:** controls how much the past hidden state influences the candidate. When $r_{t,i} \approx 0$: unit i in the candidate ignores the past state (fresh start). When $r_{t,i} \approx 1$: full past state is used (standard RNN update).

Gradient flow. The update equation $\mathbf{h}_t = (1 - \mathbf{u}_t) \odot \mathbf{h}_{t-1} + \mathbf{u}_t \odot \tilde{\mathbf{h}}_t$ creates an additive path similar to LSTM:

$$\frac{\partial h_{t,i}}{\partial h_{t-1,i}} \supseteq (1 - u_{t,i})$$

When $u_{t,i} \approx 0$, this partial derivative approaches 1, allowing gradients to flow back through time without vanishing. This is the same mechanism as the LSTM’s forget gate with $f_{t,i} \approx 1$.

Special cases.

- $\mathbf{u}_t = 1$: GRU reduces to a standard RNN with $\mathbf{h}_t = \tilde{\mathbf{h}}_t$.
- $\mathbf{u}_t = 0$: hidden state is copied unchanged, $\mathbf{h}_t = \mathbf{h}_{t-1}$ (perfect memory, no update).
- $\mathbf{r}_t = 1$: candidate uses full past state (standard update rule).
- $\mathbf{r}_t = 0$: candidate ignores past state entirely ($\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_h \mathbf{x}_t)$).

The update gate interpolates between old state (memory) and candidate (new information). The reset gate controls how much past state influences the candidate. GRU uses 25% fewer parameters than LSTM ($3d^2$ vs. $4d^2$ recurrent weight entries) with comparable performance on many sequence modelling tasks.

8.5 Bidirectional and Stacked RNNs

Bidirectional RNN. For tasks requiring access to both past and future context (e.g., named entity recognition, machine translation encoding), a standard RNN is insufficient. A Bidirectional RNN (BiRNN) runs two independent passes over the sequence:

Forward pass: $\vec{\mathbf{h}}_t = \phi(\mathbf{W}_{xh}^{\rightarrow} \mathbf{x}_t + \mathbf{W}_{hh}^{\rightarrow} \vec{\mathbf{h}}_{t-1} + \mathbf{b}^{\rightarrow})$ from $t = 1$ to T .

Backward pass: $\mathbf{h}_t = \phi(\mathbf{W}_{xh}^{\leftarrow} \mathbf{x}_t + \mathbf{W}_{hh}^{\leftarrow} \mathbf{h}_{t+1} + \mathbf{b}^{\leftarrow})$ from $t = T$ to 1.

The final representation at each step: $\mathbf{h}_t = [\vec{\mathbf{h}}_t; \mathbf{h}_t] \in \mathbb{R}^{2H}$.

This doubles the parameter count (two independent weight matrices). The full sequence must be available before the backward pass can begin — BiRNNs cannot be used for autoregressive token-by-token generation.

Stacked (Deep) RNNs. A single-layer RNN may lack the representational capacity to model complex temporal hierarchies. In a stacked RNN with D layers:

$$\mathbf{h}_t^{(d)} = \phi\left(\mathbf{W}_{xh}^{(d)} \mathbf{h}_t^{(d-1)} + \mathbf{W}_{hh}^{(d)} \mathbf{h}_{t-1}^{(d)} + \mathbf{b}^{(d)}\right), \quad d = 1, \dots, D$$

where $\mathbf{h}_t^{(0)} = \mathbf{x}_t$. Lower layers learn fine-grained local patterns; higher layers capture long-range structure. Dropout with rate p is applied to the inter-layer activations $\mathbf{h}_t^{(d-1)}$ (but not to the recurrent connection, to avoid disrupting the hidden state).

EXAMPLE — Parameter Count: 2-Layer BiLSTM

$H = 256$, $d = 300$ (word embedding), $D = 2$ layers.

- Layer 1 (bidirectional): $2 \times 4 \times (256 \cdot 300 + 256 \cdot 256 + 256) = 2 \times 4 \times 339,200 = 2,713,600$
- Layer 2 (bidirectional): $2 \times 4 \times (256 \cdot 512 + 256 \cdot 256 + 256) = 2 \times 4 \times 393,472 = 3,147,776$
- Total: $\approx 5.9\text{M}$ parameters

This architecture was standard for NLP tasks (e.g., BiDAF for reading comprehension) before the Transformer era.

8.6 LSTM Backpropagation Through Time — Gradient Analysis

The LSTM cell state update: $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t$.

The gradient of the loss with respect to $\mathbf{c}_{t-1}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{c}_{t-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{c}_t} \odot \mathbf{f}_t$$

This is element-wise multiplication by the *forget gate values* $\mathbf{f}_t \in (0, 1)^H$. Over T steps:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{c}_{t-k}} = \frac{\partial \mathcal{L}}{\partial \mathbf{c}_t} \odot \prod_{j=0}^{k-1} \mathbf{f}_{t-j}$$

Constant error carousel. If $\mathbf{f}_t \approx \mathbf{1}$ (forget gate open), the gradient flows back nearly unchanged over many time steps — this is the “constant error carousel” that allows LSTMs to learn dependencies of hundreds of time steps, impossible for vanilla RNNs.

Selective forgetting. If $\mathbf{f}_t \approx \mathbf{0}$ for some component, that dimension’s gradient is zeroed out — the LSTM chooses to forget gradients from distant past that are irrelevant.

EXAMPLE — LSTM Gradient Flow over 3 Steps

Suppose $\bar{\mathbf{c}}_3 = (1.0, 1.0)^\top$ (upstream gradient at $t = 3$). Forget gates: $\mathbf{f}_3 = (0.9, 0.1)^\top$, $\mathbf{f}_2 = (0.8, 0.9)^\top$, $\mathbf{f}_1 = (0.7, 0.2)^\top$.

$$\begin{aligned}\bar{\mathbf{c}}_2 &= (1.0 \times 0.9, 1.0 \times 0.1) = (0.90, 0.10) \\ \bar{\mathbf{c}}_1 &= (0.90 \times 0.8, 0.10 \times 0.9) = (0.72, 0.09) \\ \bar{\mathbf{c}}_0 &= (0.72 \times 0.7, 0.09 \times 0.2) = (0.504, 0.018)\end{aligned}$$

Dimension 1 retains 50.4% of the gradient over 3 steps (moderate memory). Dimension 2 retains only 1.8% (strong forgetting). The LSTM independently controls gradient flow per dimension — far more flexible than the scalar spectral radius of a plain RNN.

8.7 GRU vs. LSTM: Parameter Count and Speed

Property	LSTM	GRU
Gates	Input $\mathbf{i}$, Forget $\mathbf{f}$, Output $\mathbf{o}$, Cell $\mathbf{g}$	Reset $\mathbf{r}$, Update $\mathbf{z}$, New $\tilde{\mathbf{h}}$
State vectors	$\mathbf{h}_t, \mathbf{c}_t$ (2 per step)	$\mathbf{h}_t$ (1 per step)
Params per layer	$4H(H + d)$	$3H(H + d)$
FLOPs per step	$\sim 4H(H + d)$ MACs	$\sim 3H(H + d)$ MACs
Inference speed	Baseline	$\approx 1.25\times$ faster

GRU achieves $\approx 75\%$ of LSTM’s parameter count with comparable performance on many tasks (Chung et al., 2014). For $H = 512$, $d = 256$: LSTM has $4 \times 512 \times 768 = 1,572,864$ params per layer; GRU has 1,179,648 (saving 25%).

The Transformer

We build the Transformer from scratch: scaled dot-product attention, multi-head attention, positional encoding, and the complete Transformer block with all components in place.

9.1 The Attention Intuition

In a sequence model, the representation of token i should depend on *which other tokens are most relevant* to understanding token i . Attention computes this relevance dynamically from the data, unlike RNNs which process positions sequentially.

The problem with RNNs. In a vanilla RNN, information from token $t = 1$ must survive $T - 1$ recurrent steps to influence the representation at $t = T$. The gradient of the loss at step T with respect to the hidden state at step 1 involves a product of $T - 1$ Jacobians — causing vanishing or exploding gradients (see §8.2). LSTMs mitigate but do not eliminate this problem.

The database analogy. Attention can be understood as a differentiable database lookup:

- Each token j provides a **key** $\mathbf{k}_j$ (“what do I contain?”) and a **value** $\mathbf{v}_j$ (“what do I return?”).
- Token i issues a **query** $\mathbf{q}_i$ (“what am I looking for?”).
- The output is a weighted sum of values, with weights proportional to query-key similarity.

This is analogous to a SQL SELECT: the query selects matching keys and retrieves their values — but in a soft, differentiable way (all keys contribute, with weights given by their relevance to the query).

Why this solves the long-range problem. Attention computes a direct connection from every token i to every token j in a single operation, regardless of their distance $|i - j|$. The path length between any two positions is $O(1)$ (constant), compared to $O(T)$ for RNNs. This is the fundamental reason transformers dominate over RNNs on tasks requiring long-range dependency.

Computational cost. The direct connectivity comes at a price: attention requires $O(T^2)$ comparisons (every query against every key). For $T = 2048$ tokens: ≈ 4.2 million attention weights per head, per layer. This quadratic cost motivates the efficient attention variants in §9.6.

9.2 Self-Attention — Complete Derivation

Self-attention is the mechanism that allows each position in a sequence to gather information from all other positions in a single step, regardless of distance. This is the fundamental architectural innovation of the Transformer (Vaswani et al., 2017).

Motivation. In a sequence $\mathbf{x}_1, \dots, \mathbf{x}_T$, we want each position i to compute a context-dependent representation by attending to positions j with relevance proportional to the similarity of $\mathbf{x}_i$ and $\mathbf{x}_j$. The key insight is that “relevance” should be learned, not hand-crafted.

Query-Key-Value framework. Three learned linear projections:

$$\mathbf{q}_i = \mathbf{W}_Q \mathbf{x}_i, \quad \mathbf{k}_j = \mathbf{W}_K \mathbf{x}_j, \quad \mathbf{v}_j = \mathbf{W}_V \mathbf{x}_j$$

where $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d_k \times d}$. The query $\mathbf{q}_i$ represents “what position i is looking for”; key $\mathbf{k}_j$ represents “what position j offers”; value $\mathbf{v}_j$ represents “what position j actually contains”.

Attention weight. Compatibility between i and j :

$$e_{ij} = \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_k}}$$

Softmax normalises over all j : $\alpha_{ij} = \text{softmax}(\mathbf{e}_i)_j = e^{e_{ij}} / \sum_{j'} e^{e_{ij'}}$.

Output. Weighted sum of values:

$$\mathbf{o}_i = \sum_{j=1}^T \alpha_{ij} \mathbf{v}_j$$

Matrix form. Stack queries, keys, values as rows:

$$\mathbf{O} = \text{softmax} \left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}} \right) \mathbf{V}, \quad \mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{T \times d_k}$$

This single expression computes all T output positions simultaneously. The softmax is applied row-wise (each query independently normalised).

The self-attention mechanism is the heart of Transformers. It allows each element of a sequence to “look at” every other element in a single step, without the recurrent limitations of RNNs.

Self-Attention — Complete Derivation

Given a sequence of vectors $\mathbf{X} \in \mathbb{R}^{T \times d}$ (where T is the length and d the dimension):

Linear projections:

$$\mathbf{Q} = \mathbf{XW}^Q, \quad \mathbf{K} = \mathbf{XW}^K, \quad \mathbf{V} = \mathbf{XW}^V$$

with $\mathbf{W}^Q, \mathbf{W}^K \in \mathbb{R}^{d \times d_k}, \mathbf{W}^V \in \mathbb{R}^{d \times d_v}$.

Attention matrix:

$$\mathbf{A} = \text{softmax} \left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}} \right) \in \mathbb{R}^{T \times T}$$

A_{ij} is the attention weight of token i toward token j .

Output:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{AV} \in \mathbb{R}^{T \times d_v}$$

Why the $1/\sqrt{d_k}$ factor. The dot product $\mathbf{QK}^\top$ grows in magnitude with d_k : if Q_i and K_j are i.i.d. with mean 0 and variance 1, then $Q_i \cdot K_j = \sum_{l=1}^{d_k} Q_{il} K_{jl}$ has variance d_k . Dividing by $\sqrt{d_k}$ stabilises the variance to 1, preventing the softmax from saturating in near-zero-gradient regions.

Q, K, V interpretation. Borrowed from information retrieval systems:

- **Q** (query): “what am I looking for?”

- **K** (key): “what can I offer?”
- **V** (value): “what is my informational content?”

A_{ij} measures the compatibility between the query of i and the key of j ; the output is the weighted average of the values.

EXAMPLE — Self-Attention on a Short Sequence ($T = 3, d_k = 2$)

$$\mathbf{Q} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{K} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}$$

$$\mathbf{QK}^\top = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 2 & 1 \\ 2 & 3 & 1 \end{pmatrix}, \quad \frac{\mathbf{QK}^\top}{\sqrt{2}} \approx \begin{pmatrix} 0.71 & 0.71 & 0 \\ 0.71 & 1.41 & 0.71 \\ 1.41 & 2.12 & 0.71 \end{pmatrix}$$

After row-wise softmax, we obtain a probability distribution indicating which tokens each position attends to most.

Computational complexity. Self-attention requires $O(T^2d)$ for the product $\mathbf{QK}^\top$: quadratic in sequence length. For $T = 4096, d = 512$: $\approx 8.6 \times 10^9$ operations per layer. This is the bottleneck for long sequences, motivating variants such as Linformer, Performer, and FlashAttention (Volume V).

Causal masking. In the autoregressive decoder, the token at position t must not “see” future positions. A causal mask is applied:

$$\tilde{A}_{ij} = \begin{cases} A_{ij} & j \leq i \\ -\infty & j > i \end{cases}$$

After softmax, masked positions have weight 0 (since $e^{-\infty} = 0$).

9.3 Multi-Head Attention — Why Multiple Heads

Single-head attention projects queries, keys, and values into a single representation space. This limits the model’s ability to simultaneously focus on different types of relationships (syntax, semantics, coreference). Multi-Head Attention resolves this by running H attention operations in parallel over independent subspaces.

Multi-Head Attention

With H heads and $d_k = d_v = d/H$:

$$\text{head}_h = \text{Attention}(\mathbf{XW}_h^Q, \mathbf{XW}_h^K, \mathbf{XW}_h^V), \quad h = 1, \dots, H$$

$$\text{MultiHead}(\mathbf{X}) = \text{Concat}(\text{head}_1, \dots, \text{head}_H) \mathbf{W}^O$$

with $\mathbf{W}_h^Q, \mathbf{W}_h^K \in \mathbb{R}^{d \times d_k}, \mathbf{W}_h^V \in \mathbb{R}^{d \times d_v}, \mathbf{W}^O \in \mathbb{R}^{Hd_v \times d}$.

Why it works better. Each head can specialise in a different type of relationship. Empirical analyses of BERT (Clark et al., 2019) show that different heads capture: (i) local syntactic dependencies, (ii) long-range coreference, (iii) semantic relations, (iv) positional patterns. A single attention head would need to compress all this information into one space, creating interference.

MHA parameters. With H heads, $d_k = d_v = d/H$:

- $\mathbf{W}_h^Q, \mathbf{W}_h^K, \mathbf{W}_h^V$: $3H \times d \times (d/H) = 3d^2$ parameters in total.
- $\mathbf{W}^O$: $d \times d$ parameters.
- Total: $4d^2$ parameters (the same count as single-head attention with $d_k = d$!).

The computational efficiency is identical, but the expressive capacity is greater.

EXAMPLE — MHA with $H = 2, d = 4, d_k = 2$

The two heads project the input into two different 2D subspaces. Head 1 might focus on nearby positions (short-range), head 2 on distant ones (long-range). The concatenation $[h_1; h_2] \in \mathbb{R}^{T \times 4}$ is re-projected by $\mathbf{W}^O \in \mathbb{R}^{4 \times 4}$ to blend the two perspectives into a unified output.

Efficient implementation. In practice, the H heads are not computed sequentially but in parallel by reshaping tensors:

1. $\mathbf{Q} = \mathbf{XW}^Q \in \mathbb{R}^{T \times d} \rightarrow$ reshape to $\mathbb{R}^{T \times H \times d_k} \rightarrow$ permute to $\mathbb{R}^{H \times T \times d_k}$
2. Compute attention in parallel across all heads: $\mathbb{R}^{H \times T \times T}$
3. Multiply by $\mathbf{V}$: $\mathbb{R}^{H \times T \times d_v} \rightarrow$ permute and reshape to $\mathbb{R}^{T \times d}$

This implementation fully exploits GPU parallelism.

9.4 Positional Encoding

Self-attention is *permutation equivariant*: if we permute the input sequence, the output is permuted the same way. This is desirable for set-valued inputs but wrong for sequential data where order matters. Positional encoding breaks this symmetry by injecting position information.

Sinusoidal encoding. Vaswani et al. (2017) define:

$$\text{PE}(t, 2k) = \sin\left(\frac{t}{10000^{2k/d}}\right), \quad \text{PE}(t, 2k+1) = \cos\left(\frac{t}{10000^{2k/d}}\right)$$

for position $t \in \{1, \dots, T\}$ and dimension index $k \in \{0, \dots, d/2 - 1\}$.

Geometric interpretation. Each pair of dimensions $(2k, 2k+1)$ encodes position as a 2D unit vector rotating at frequency $\omega_k = 10000^{-2k/d}$:

$$\text{PE}(t, 2k : 2k+2) = (\sin \omega_k t, \cos \omega_k t)$$

The frequencies range from $\omega_0 = 1$ (one full rotation per $2\pi \approx 6.3$ positions) to $\omega_{d/2-1} = 10000^{-1}$ (one rotation per $\approx 62,832$ positions). Different dimensions encode position at different scales, like a Fourier basis.

Relative position property. There exists a linear transformation $\mathbf{M}_\delta$ such that $\text{PE}(t + \delta) = \mathbf{M}_\delta \text{PE}(t)$ for all t, δ . Proof: using the angle addition formulas $\sin(a + b) = \sin a \cos b + \cos a \sin b$ and $\cos(a + b) = \cos a \cos b - \sin a \sin b$:

$$\mathbf{M}_\delta = \bigoplus_{k=0}^{d/2-1} \begin{pmatrix} \cos \omega_k \delta & \sin \omega_k \delta \\ -\sin \omega_k \delta & \cos \omega_k \delta \end{pmatrix}$$

This means the attention mechanism can attend to relative positions by linearly transforming positional encodings.

EXAMPLE — Positional Encoding Values, $d = 4$, Positions 1–4

$$\omega_0 = 1, \omega_1 = 10000^{-1/2} \approx 0.01.$$

t	$\sin(\omega_0 t)$	$\cos(\omega_0 t)$	$\sin(\omega_1 t)$	$\cos(\omega_1 t)$
1	0.8415	0.5403	0.0100	0.9999
2	0.9093	-0.4161	0.0200	0.9998
3	0.1411	-0.9900	0.0300	0.9996
4	-0.7568	-0.6536	0.0400	0.9992

The fast dimensions (ω_0) change rapidly, distinguishing nearby positions. The slow dimensions (ω_1) change gradually, distinguishing distant positions. Together they form a unique “fingerprint” for each position up to $T = 62,832$.

Self-attention is *permutation-equivariant*: shuffling the input rows of $\mathbf{X}$ just shuffles the output rows — no positional information is captured. We inject position via sinusoidal encoding:

$$\text{PE}_{(t,2k)} = \sin\left(\frac{t}{10000^{2k/d}}\right), \quad \text{PE}_{(t,2k+1)} = \cos\left(\frac{t}{10000^{2k/d}}\right)$$

Added to the embedding: $\mathbf{x}_t \leftarrow \mathbf{x}_t + \text{PE}_t$.

Why sinusoids? (1) The dot product $\text{PE}_t \cdot \text{PE}_{t+\Delta}$ depends only on Δ (relative offset), not on absolute positions. (2) Can extrapolate to sequence lengths unseen during training.

9.5 The Complete Transformer Block

Each Transformer layer consists of:

$$\mathbf{X}' = \text{LayerNorm}(\mathbf{X} + \text{MultiHead}(\mathbf{X}))$$

$$\mathbf{X}'' = \text{LayerNorm}(\mathbf{X}' + \text{FFN}(\mathbf{X}'))$$

where $\text{FFN}(\mathbf{x}) = \mathbf{W}_2 \text{ReLU}(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$ with inner dimension $d_{\text{ff}} = 4d$.

Layer Normalization. Unlike BatchNorm, LayerNorm normalises over the feature dimension for a single example:

$$\text{LayerNorm}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \mu_x}{\sigma_x + \epsilon} + \beta$$

$\mu_x = \frac{1}{d} \sum_i x_i, \sigma_x^2 = \frac{1}{d} \sum_i (x_i - \mu_x)^2$. Suitable for variable-length sequences.

EXAMPLE — Attention score computation ($T = 3, d_k = 2$)

$$\mathbf{Q} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{K} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}$$

$$\mathbf{S} = \mathbf{QK}^\top = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 2 \end{pmatrix}$$

$$\mathbf{S}/\sqrt{2} = \begin{pmatrix} 0.707 & 0 & 0.707 \\ 0 & 0.707 & 0.707 \\ 0.707 & 0.707 & 1.414 \end{pmatrix}$$

Row 3: $\text{softmax}(0.707, 0.707, 1.414)$. With $e^{0.707} \approx 2.028$ and $e^{1.414} \approx 4.113$:

$$A_{3.} = \frac{(2.028, 2.028, 4.113)}{8.169} \approx (0.248, 0.248, 0.504)$$

Token 3 attends mostly to itself (0.504) and equally to tokens 1 and 2.

9.6 Efficient Attention Variants

Standard attention. Time $O(T^2d)$, memory $O(T^2)$ — quadratic in sequence length. For $T = 32,768$ (LLaMA-3 context) and $d = 128$ (head dim): $T^2 = 10^9$ operations per layer — borderline feasible on A100 but prohibitive for $T > 10^5$.

Sparse attention. Rather than attending to all T positions, each query attends to a fixed set $S_i \subset \{1, \dots, T\}$ of $k \ll T$ positions. Complexity $O(Tkd)$. Examples:

- *Local (sliding window)*: each token attends to k nearest neighbours. Good for tasks with local structure (text, images). Used in Longformer and BigBird.
- *Strided*: attend to every s -th token for long-range dependencies.
- *Global tokens*: a few tokens (e.g. [CLS]) attend to all positions; all positions attend to them. Allows $O(T)$ global information exchange.

Linear attention. Replace softmax with a kernel $\kappa(\mathbf{q}, \mathbf{k}) = \phi(\mathbf{q})^\top \phi(\mathbf{k})$ where $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^r$ is a feature map. Then:

$$\mathbf{o}_i = \frac{\phi(\mathbf{q}_i)^\top \left(\sum_j \phi(\mathbf{k}_j) \mathbf{v}_j^\top \right)}{\phi(\mathbf{q}_i)^\top \sum_j \phi(\mathbf{k}_j)}$$

Precompute $\mathbf{S} = \sum_j \phi(\mathbf{k}_j) \mathbf{v}_j^\top \in \mathbb{R}^{r \times d}$ and $\mathbf{z} = \sum_j \phi(\mathbf{k}_j) \in \mathbb{R}^r$ once: total cost $O(Trd)$ — linear in T . The trade-off is that removing softmax's normalisation reduces expressivity: linear attention is weaker at sharp, selective retrieval.

Performer (Choromanski et al., 2021). Uses random Fourier features to approximate the softmax kernel: $\phi(\mathbf{x}) = \frac{1}{\sqrt{r}} [\cos(\square_1^\top \mathbf{x}), \dots, \sin(\square_r^\top \mathbf{x})]$, $\square_i \sim \mathcal{N}(0, \mathbf{I})$. Provides an unbiased estimator of $\exp(\mathbf{q}^\top \mathbf{k} / \sqrt{d})$ with $O(rd)$ per token.

FlashAttention (Dao et al., 2022). Does *not* change the mathematical result — it computes exact softmax attention. Instead, it tiles the attention computation to fit in GPU SRAM (fast memory, ~ 20 MB), avoiding writing the $T \times T$ matrix to slow HBM (~ 40 – 80 GB). IO complexity:

$$\text{Standard: } O(T^2) \text{ HBM accesses; } \quad \text{FlashAttention: } O(T^2/M) \text{ HBM accesses}$$

where M is SRAM size. Speedup 2–4 $\times$ in wall-clock time on A100 GPUs. FlashAttention-2 adds better parallelism over sequence length. FlashAttention-3 (2024) exploits hardware-specific features of H100 (warp specialisation, TMA). These are critical for training LLMs at scale.

Method	Time	Memory	Exact?	Use case
Standard	$O(T^2d)$	$O(T^2)$	Yes	$T \leq 4096$
FlashAttention	$O(T^2d)$	$O(T)$	Yes	$T \leq 128K$
Sparse	$O(Tkd)$	$O(Tk)$	No	Long docs
Linear	$O(Trd)$	$O(r)$	Approx	Very long seqs

9.7 Relative and ALiBi Positional Encoding

Absolute sinusoidal or learned positional encodings add position information to token embeddings before the attention layers. An alternative is to encode position *directly* in the attention computation.

Relative PE (Shaw et al., 2018). Replace the standard attention logit:

$$e_{ij} = \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_k}} \rightarrow e_{ij} = \frac{(\mathbf{q}_i + \mathbf{a}_{ij}^Q)^\top (\mathbf{k}_j + \mathbf{a}_{ij}^K)}{\sqrt{d_k}}$$

where $\mathbf{a}_{ij}^Q, \mathbf{a}_{ij}^K \in \mathbb{R}^{d_k}$ are learned embeddings that depend only on the relative offset $i - j$ (clipped to a maximum distance k : $\text{clip}(i - j, -k, k)$). This allows the model to distinguish “ j is 3 tokens before i ” from “ j is 5 tokens after i ” regardless of absolute position.

Comparison of absolute vs. relative PE:

Method	Extra parameters	Length generalisation
Sinusoidal (absolute)	0	Limited
Learned absolute	$T_{\max} \times d$	Poor (fixed $T_{\max}$)
Relative PE	$2k \times d_k \times 2$	Good
RoPE (§9.9)	0	Good
ALiBi	H slopes	Excellent

ALiBi (Attention with Linear Biases; Press et al., 2022). Add a head-specific linear bias to the attention logits:

$$e_{ij} = \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_k}} - m_h \cdot (i - j), \quad i \geq j$$

where $m_h > 0$ is a fixed (non-learned) slope for head h : $m_h = 2^{-h \cdot 8/H}$. The penalty $m_h(i - j)$ grows linearly with distance, so attending to distant tokens is progressively penalised.

Why ALiBi generalises to long sequences. A model trained on context $T_{\text{train}} = 1024$ can still process $T = 2048$ tokens at inference: the distance penalty already exists as an inductive bias at any distance, so the model has implicitly learned to discount distant tokens. This *length extrapolation* without fine-tuning is a major practical advantage. ALiBi is used in BLOOM (176B parameter model).

9.8 Flash Attention: IO-Aware Self-Attention

Standard attention materialises the full $T \times T$ attention matrix $\mathbf{A} = \text{softmax}(\mathbf{Q}\mathbf{K}^\top / \sqrt{d_k}) \in \mathbb{R}^{T \times T}$ in GPU HBM (high-bandwidth memory). For $T = 4096$, $d_k = 64$: $\mathbf{A}$ is $4096^2 \times 4 = 67$ MB per head — and the memory bandwidth required to read/write it dominates compute time.

Flash Attention (Dao et al., 2022) avoids materialising $\mathbf{A}$ by computing attention in tiles that fit in SRAM (fast on-chip memory):

1. Partition $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ into blocks of size $B_r \times d, B_c \times d$.
2. For each query block, iterate over all key/value blocks: compute local softmax numerators and running softmax denominators, updating the output incrementally.
3. The final output $\mathbf{O}$ is assembled without ever storing the full $\mathbf{A}$.

Complexity.

	Standard Attention	Flash Attention
Memory	$O(T^2)$	$O(T)$
FLOPs	$O(T^2 d)$	$O(T^2 d)$ (same!)
HBM reads/writes	$O(T^2)$	$O(T^2 d/M)$ (M = SRAM size)
Speed (A100 GPU)	Baseline	2–4× faster

Flash Attention does not reduce FLOPs; it reduces memory bandwidth, which is the true bottleneck. This is a general principle in modern ML: *compute is cheap, memory bandwidth is expensive*.

9.9 Rotary Position Embeddings (RoPE)

Sinusoidal position encodings (§9.4) are additive and do not interact with attention scores in a position-relative way. RoPE (Su et al., 2021) instead rotates query and key vectors in 2D subspaces, encoding position as a rotation angle.

Definition. For position m , the RoPE rotation on the k -th 2D subspace (dimensions $2k - 1$ and $2k$):

$$\mathbf{q}_m^{(k)} = \begin{pmatrix} \cos m\theta_k & -\sin m\theta_k \\ \sin m\theta_k & \cos m\theta_k \end{pmatrix} \begin{pmatrix} q_{2k-1} \\ q_{2k} \end{pmatrix}, \quad \theta_k = 10000^{-2(k-1)/d}$$

Key property. The dot product between rotated query at position m and key at position n :

$$\langle \mathbf{q}_m, \mathbf{k}_n \rangle = f(\mathbf{q}, \mathbf{k}, m - n)$$

The attention score depends only on the *relative position* $m - n$, not absolute positions. This gives RoPE natural length generalisation: a model trained on $T = 2048$ tokens can attend to relative positions beyond T without retraining.

Adoption. RoPE is used in LLaMA 1/2/3, Mistral, Gemma, Qwen, and most modern open-source LLMs. It has largely replaced learned and sinusoidal absolute encodings.

Sequence-to-Sequence and Cross-Attention

Encoder–decoder architectures, cross-attention mechanism, and the complete Transformer for sequence transduction.

10.1 Encoder–Decoder Architecture

In sequence-to-sequence tasks (translation, summarisation, ASR), we want to map an input sequence $\mathbf{x}_{1:T}$ to an output sequence $\mathbf{y}_{1:S}$ of (possibly) different length.

The **encoder** applies N_e Transformer blocks to $\mathbf{x}_{1:T}$, producing context representations $\mathbf{z}_{1:T} = \text{Enc}(\mathbf{x}_{1:T})$.

The **decoder** generates $\mathbf{y}_t$ autoregressively, attending to both previous outputs (self-attention) and the encoder output (cross-attention).

Why separate encoder and decoder? In machine translation, the source language (encoder input) and target language (decoder output) have different lengths and word orders. A single autoregressive model (decoder-only) can handle this but must carry all source context in a single hidden vector, creating an information bottleneck. The encoder-decoder architecture removes this bottleneck: the decoder has direct (via cross-attention) access to *all* source positions at every generation step.

Encoder stack. Each encoder block applies:

$$\mathbf{H}^{(l)} = \text{LayerNorm}(\mathbf{H}^{(l-1)} + \text{MultiHeadSelfAttn}(\mathbf{H}^{(l-1)}))$$

$$\mathbf{H}^{(l)} \leftarrow \text{LayerNorm}(\mathbf{H}^{(l)} + \text{FFN}(\mathbf{H}^{(l)}))$$

Crucially, encoder self-attention is *bidirectional*: every source token can attend to every other source token. This allows the encoder to build contextual representations incorporating both left and right context.

Decoder stack. Each decoder block applies three sublayers:

1. **Masked self-attention**: decoder tokens attend to previous decoder tokens only (causal mask ensures autoregressive generation).
2. **Cross-attention**: decoder queries attend to all encoder outputs (§10.2).
3. **FFN**: position-wise transformation.

Parameter count. For encoder depth N_e , decoder depth N_d , model dimension d , FFN width d_{ff} , heads H :

$$P_{\text{enc}} = N_e(4d^2 + 2dd_{\text{ff}}), \quad P_{\text{dec}} = N_d(6d^2 + 2dd_{\text{ff}})$$

(Decoder has 3 attention sublayers vs. 2 for encoder, hence $6d^2$ vs. $4d^2$.)

For the original Transformer ($N_e = N_d = 6$, $d = 512$, $d_{\text{ff}} = 2048$, $H = 8$):

$$P_{\text{enc}} = 6(4 \times 512^2 + 2 \times 512 \times 2048) = 6(1.05M + 2.1M) = 18.9M$$

$$P_{\text{dec}} = 6(6 \times 512^2 + 2 \times 512 \times 2048) = 6(1.57M + 2.1M) = 22.0M$$

Total (excluding embedding): $\approx 41M$ parameters.

EXAMPLE — T5 vs. BERT: encoder-only vs. encoder-decoder

BERT-base (encoder-only, $N_e = 12$, $d = 768$): $\approx 110M$ parameters. Bidirectional encoder, no decoder. Used for classification, NER, QA (non-generative).

T5-base (encoder-decoder, $N_e = N_d = 12$, $d = 768$): $\approx 220M$ parameters. Frames all NLP tasks as text-to-text: “translate English to German: The house is wonderful.” $\rightarrow$ “Das Haus ist wunderbar.”

GPT-2 (decoder-only, 12 layers): $\approx 117M$ parameters. Autoregressive left-to-right; no cross-attention. Efficient for generation but cannot condition on a full source sequence without including it in the context window.

10.2 Cross-Attention

In an encoder-decoder transformer, the decoder must attend to the encoder’s output (the source context). This is achieved via **cross-attention**: the queries come from the decoder’s current hidden state, while the keys and values come from the encoder output.

Mechanism. Let $\mathbf{E} \in \mathbb{R}^{T_s \times d}$ be the encoder output (source sequence of length T_s) and $\mathbf{D} \in \mathbb{R}^{T_t \times d}$ be the decoder state (target sequence of length T_t). Cross-attention:

$$\mathbf{Q} = \mathbf{D}\mathbf{W}^Q, \quad \mathbf{K} = \mathbf{E}\mathbf{W}^K, \quad \mathbf{V} = \mathbf{E}\mathbf{W}^V$$

$$\text{CrossAttn}(\mathbf{D}, \mathbf{E}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right) \mathbf{V}$$

The attention matrix $\mathbf{A} \in \mathbb{R}^{T_t \times T_s}$ has entry A_{ij} representing how much decoder position i attends to encoder position j .

Differences from self-attention.

- Queries: from the decoder (target side).
- Keys and values: from the encoder (source side), same for all decoder layers.
- No causal mask: each decoder position can attend to any encoder position.
- Shapes: $\mathbf{Q} \in \mathbb{R}^{T_t \times d_k}$, $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{T_s \times d_k}$, output $\in \mathbb{R}^{T_t \times d}$.

Alignment. The cross-attention weights A_{ij} learn source-target alignment: in machine translation, French word i attends strongly to the corresponding English word(s) j . Visualising these weights was one of the first interpretability results in neural MT (Bahdanau et al., 2015).

In the decoder, each layer applies:

Self-Attention block: $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ all from decoder hidden states

Cross-Attention block: $\mathbf{Q}$ from decoder, $\mathbf{K}, \mathbf{V}$ from encoder output $\mathbf{z}_{1:T}$

$$\text{CrossAttn}(\mathbf{Q}_{\text{dec}}, \mathbf{Z}) = \text{softmax}\left(\frac{\mathbf{Q}_{\text{dec}}\mathbf{Z}^\top\mathbf{W}^K}{\sqrt{d_k}}\right) \mathbf{Z}\mathbf{W}^V$$

Each decoder position can attend to all encoder positions. The softmax weights A_{tj} tell us: when predicting output token at position t , how much should we look at input position j ?

10.3 Complete Decoder Block

$$\mathbf{H}' = \text{LayerNorm}(\mathbf{H} + \text{MaskedSelfAttn}(\mathbf{H}))$$

$$\mathbf{H}'' = \text{LayerNorm}(\mathbf{H}' + \text{CrossAttn}(\mathbf{H}', \mathbf{Z}))$$

$$\mathbf{H}''' = \text{LayerNorm}(\mathbf{H}'' + \text{FFN}(\mathbf{H}''))$$

Each decoder block contains three sub-layers, each wrapped with a residual connection and layer normalisation. Let T_{dec} be the current output length and T_{enc} be the encoder sequence length, with hidden dimension d .

Sub-layer 1: Masked Causal Self-Attention. Queries, keys, and values all come from the decoder hidden state $\mathbf{H} \in \mathbb{R}^{T_{\text{dec}} \times d}$. The attention mask $\mathbf{M} \in \{0, -\infty\}^{T_{\text{dec}} \times T_{\text{dec}}}$ is lower-triangular:

$$M_{ij} = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases}$$

Adding $\mathbf{M}$ to the pre-softmax attention scores zeros out future positions, enforcing autoregressive generation: position i can only attend to positions $j \leq i$.

Sub-layer 2: Cross-Attention. Queries come from the decoder state $\mathbf{H}' \in \mathbb{R}^{T_{\text{dec}} \times d}$; keys and values come from the encoder output $\mathbf{Z} \in \mathbb{R}^{T_{\text{enc}} \times d}$:

$$\mathbf{Q} = \mathbf{H}'\mathbf{W}^Q, \quad \mathbf{K} = \mathbf{Z}\mathbf{W}^K, \quad \mathbf{V} = \mathbf{Z}\mathbf{W}^V$$

This allows each decoder position to attend over the full encoder representation without future masking (the encoder output is fully observed). The attention scores matrix $\in \mathbb{R}^{T_{\text{dec}} \times T_{\text{enc}}}$ — not square if source and target lengths differ.

Sub-layer 3: Feed-Forward Network. Identical to the encoder FFN: two linear layers with a ReLU/GELU activation, expanding to $4d$ and projecting back to d .

Parameter count per decoder block. With $d = 512$, $h = 8$ heads, $d_{\text{ff}} = 2048$:

- Masked self-attention: $4d^2 = 4 \times 512^2 = 1.05\text{M}$ parameters.
- Cross-attention: $4d^2 = 1.05\text{M}$ (separate $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V, \mathbf{W}^O$).
- FFN: $2 \times d \times 4d = 2 \times 512 \times 2048 = 2.1\text{M}$ parameters.
- Layer norms: $3 \times 2d = 3072$ parameters (negligible).
- Total per block: $\approx 4.2\text{M}$ parameters.

A 6-layer decoder has $\approx 25\text{M}$ parameters, comparable to the encoder.

10.4 Generation Strategies

Given a decoder-only or encoder-decoder model, generating text requires producing a sequence of tokens $y_1, y_2, \dots, y_T$ one at a time. The model outputs a probability distribution over the vocabulary $p(y_t | y_1, \dots, y_{t-1})$ at each step. Different generation strategies make different trade-offs between quality and diversity.

Greedy decoding.

$$y_t = \arg \max_v p(y_t = v | y_{<t})$$

Fast (no search), deterministic. Tends to produce repetitive, degenerate outputs: once the model produces “the the the...” it may continue indefinitely since the greedy choice at each step is the same word.

Beam search. Maintain the k highest-probability partial sequences (“beams”) at each step. At step t , expand each beam into $|V|$ continuations and keep the top k by cumulative log-probability:

$$\text{score}(y_{1:t}) = \sum_{i=1}^t \log p(y_i | y_{<i})$$

Cost: $k \times |V|$ evaluations per step. Standard in machine translation ($k = 4$ or $k = 5$). Length penalty applied to avoid bias toward short sequences: score/T^α for $\alpha \in [0.6, 0.8]$.

Limitations of beam search. Maximising likelihood does not produce the highest-quality text. Holtzman et al. (2020) showed that beam-search text is often dull and formulaic; humans prefer stochastic outputs even when they have lower likelihood. This motivates sampling-based strategies (§10.5).

EXAMPLE — Beam Search Step, $k = 2$, $|V| = 5$

At $t = 2$, beams: {“The cat”: -0.8 , “A dog”: -1.2 }.
 Expand “The cat”: top continuations “sat” (-1.3), “ran” (-1.6), “meowed” (-2.1).
 Expand “A dog”: top continuations “barked” (-1.9), “ran” (-2.0).
 All candidates: {“The cat sat”: -1.3 , “The cat ran”: -1.6 , “A dog barked”: -1.9 , “A dog ran”: -2.0 , “The cat meowed”: -2.1 }.
 Keep top $k = 2$: “The cat sat” (-1.3) and “The cat ran” (-1.6).

Greedy: $y_t = \arg \max_k \hat{y}_{t,k}$. Fast but locally optimal.

Beam search: maintain B candidate sequences; expand each by top- B tokens; keep top- B . Returns globally better sequences.

Nucleus (top- p) sampling: sum sorted probabilities until cumulative $\geq p$; sample from this dynamic vocabulary. Balances diversity and quality. Typical: $p = 0.9$, $T = 0.7$.

Perplexity:

$$\text{PPL} = \exp \left(-\frac{1}{T} \sum_{t=1}^T \log p(y_t | y_{<t}) \right)$$

Perplexity 100 means the model is on average as uncertain as a uniform distribution over 100 tokens.

10.5 Sampling Strategies for Text Generation

Given a language model’s next-token distribution $P(\text{token} \mid \text{context})$, different sampling strategies trade diversity for coherence.

Greedy decoding. Select $\hat{w}_t = \arg \max_w P(w \mid \text{context})$ at each step. Fast, deterministic, but often repetitive and suboptimal: the globally most likely sequence is not the sequence of locally most likely tokens.

Beam search. Maintain k candidate sequences (“beams”) at each step. At time t , expand each beam by all vocabulary items, keep the top k by cumulative log-probability. Final output: highest-scoring beam.

EXAMPLE — Beam Search with $k = 2$, Vocabulary $\{A, B, C\}$

Step 0: beams = $\{[]\}$ (empty).

Step 1: expand to $\{[A], [B], [C]\}$ with log-probs $\{-0.5, -1.2, -2.0\}$. Keep top 2: $\{[A], [B]\}$.

Step 2: expand $[A] \rightarrow \{[AA], [AB], [AC]\}$ with log-probs $\{-1.0, -1.8, -3.5\}$; expand $[B] \rightarrow \{[BA], [BB], [BC]\}$ with $\{-2.5, -2.6, -4.2\}$. Top 2 overall: $[AA]$ (-1.0), $[AB]$ (-1.8).

Greedy would have selected $[A, A]$ anyway here, but in general beam search recovers lower-probability completions that lead to higher-probability full sequences.

Temperature scaling. Divide logits by $T > 0$ before softmax: $P_T(w) \propto \exp(z_w/T)$.

- $T \rightarrow 0$: approaches greedy (spiky distribution).
- $T = 1$: original model distribution.
- $T > 1$: flatter distribution, more diversity.

Top- p (nucleus) sampling. Sort tokens by probability. Include the smallest set S such that $\sum_{w \in S} P(w) \geq p$. Sample from S with renormalised probabilities. Typical $p = 0.9$. Adapts dynamically: sharp distributions (clear next token) use few candidates; flat distributions use many.

Top- k sampling. Keep only the k most probable tokens, renormalise. Fixed $k = 50$ or 100 is less adaptive but simpler.

10.6 Encoder–Decoder vs. Decoder-Only Architecture

Property	Encoder-Decoder (T5, BART)	Decoder-Only (GPT, LLaMA)
Input processing	Bidirectional encoder	Causal (left-to-right)
Cross-attention	Yes (decoder attends to encoder)	No
Suitable for	Seq2seq (translation, summarisation)	Generation, in-context learning
Param efficiency	Lower (two stacks)	Higher (one stack)
Scaling trend	Less common at scale	Dominant (2022+)

Why decoder-only dominates. Training objective: predict next token (causal language modelling). With decoder-only, the same forward pass computes predictions for all positions simultaneously (teacher forcing), making training efficient. Cross-attention in encoder-decoder doubles the KV-cache overhead at inference. Scaling laws (Hoffmann et al., 2022) favour the simpler architecture.

Practical Training Recipes

Weight initialisation, learning rate scheduling, and the complete training loop — every hyperparameter decision explained.

11.1 Weight Initialisation

Proper initialisation ensures that (a) activations do not saturate at initialisation, (b) gradients do not vanish or explode at step 0, and (c) the scale of updates is consistent across layers from the first step.

Why initialisation matters. Consider a deep network with L layers, each with n neurons, and weight matrices sampled i.i.d. with $W_{ij} \sim \mathcal{N}(0, \sigma^2)$. The pre-activation at layer l :

$$\text{Var}[z_j^{(l)}] = n\sigma^2 \cdot \text{Var}[a_j^{(l-1)}]$$

If $n\sigma^2 \neq 1$: the variance multiplies by $n\sigma^2$ at each layer, so $\text{Var}[z^{(L)}] = (n\sigma^2)^L \text{Var}[z^{(0)}]$. For $n = 500, \sigma^2 = 0.01, L = 10$: $(5.0)^{10} \approx 10^7$ — catastrophic explosion.

LeCun initialisation (for tanh). Near $z = 0$, $\tanh(z) \approx z$, so $\text{Var}[\tanh(z)] \approx \text{Var}[z]$. Requirement: $n\sigma^2 = 1$, giving:

$$\sigma^2 = \frac{1}{n_{\text{in}}} \quad (\text{LeCun, 1998})$$

Xavier/Glorot initialisation (for tanh/sigmoid). Accounts for both forward and backward variance preservation:

$$\sigma^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}} \quad (\text{Glorot \& Bengio, 2010})$$

He initialisation (for ReLU). ReLU kills half the neurons ($z < 0$), effectively halving the variance:

$$\text{Var}[\text{ReLU}(z)] \approx \frac{1}{2} \text{Var}[z]$$

Compensating: $\frac{1}{2}n_{\text{in}}\sigma^2 = 1$, giving:

$$\sigma^2 = \frac{2}{n_{\text{in}}} \quad (\text{He et al., 2015})$$

PyTorch uses He initialisation for linear and convolutional layers by default (`torch.nn.init.kaiming_normal_ with mode='fan_in'`). For transformers, GPT-2 scales the residual branch initialisation by $1/\sqrt{2L}$ (where L = number of layers) to prevent the residual stream from growing with depth.

Poor initialisation causes activations to explode or vanish before training begins.

Xavier / Glorot initialisation (for tanh/sigmoid activations):

$$W_{ij} \sim \mathcal{U} \left[-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}} \right]$$

Derivation. Require $\text{Var}(\mathbf{z}^{(l)}) = \text{Var}(\mathbf{z}^{(l-1)})$ (forward pass) and $\text{Var}(\mathbf{a}^{(l-1)}) = \text{Var}(\mathbf{a}^{(l)})$ (backward pass). Under the approximation $\phi'(z) \approx 1$ (valid near 0 for tanh):

Forward constraint: $\text{Var}(z_i) = n_{\text{in}} \text{Var}(w) \text{Var}(a) \Rightarrow \text{Var}(w) = 1/n_{\text{in}}$

Backward constraint: $\text{Var}(w) = 1/n_{\text{out}}$

Glorot compromises: $\text{Var}(w) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$.

He / Kaiming initialisation (for ReLU activations):

$$W_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right)$$

The factor 2 compensates for ReLU zeroing half the activations: $\mathbb{E}[\text{ReLU}(z)^2] = \frac{1}{2}\mathbb{E}[z^2]$ when z is symmetric around 0.

11.2 Learning Rate Scheduling

A fixed learning rate α is suboptimal: large α is needed early (fast exploration) but causes instability near the optimum; small α is needed late (precise convergence) but slows early training. Learning rate schedules adapt α over training.

Linear warmup. Start from $\alpha_0 \approx 0$ and linearly increase to the peak α_{max} over T_{warm} steps:

$$\alpha_t = \alpha_{\text{max}} \cdot \frac{t}{T_{\text{warm}}}, \quad t \leq T_{\text{warm}}$$

Warmup stabilises training at initialisation, when parameters are random and gradients are noisy. Without warmup, Adam's second-moment estimates $\hat{v}$ start at 0 and underestimate gradient variance, causing large initial steps. Warmup period: typically $T_{\text{warm}} = 4\% - 6\%$ of total training steps.

Cosine annealing. After warmup, decay with a cosine schedule:

$$\alpha_t = \alpha_{\text{min}} + \frac{1}{2}(\alpha_{\text{max}} - \alpha_{\text{min}}) \left(1 + \cos\left(\pi \cdot \frac{t - T_{\text{warm}}}{T_{\text{total}} - T_{\text{warm}}}\right)\right)$$

Decreases smoothly from α_{max} to α_{min} (typically $0.01\alpha_{\text{max}}$). Empirically outperforms step decay and linear decay on large-scale training runs (e.g., GPT-3, LLaMA use cosine).

Transformer schedule (Vaswani et al., 2017):

$$\alpha_t = d_{\text{model}}^{-0.5} \cdot \min(t^{-0.5}, t \cdot T_{\text{warm}}^{-1.5})$$

Increases for $t < T_{\text{warm}}$ and decays as $t^{-0.5}$ thereafter. The $d_{\text{model}}^{-0.5}$ factor normalises for model width.

EXAMPLE — Schedule Comparison, $T = 100\text{k steps}$, $\alpha_{\text{max}} = 3 \times 10^{-4}$

Step	Warmup (linear)	Cosine	Step decay	Constant
1k (1%)	3×10^{-5}	3×10^{-5}	3×10^{-4}	3×10^{-4}
5k (5%)	1.5×10^{-4}	1.5×10^{-4}	3×10^{-4}	3×10^{-4}
50k (50%)	3×10^{-4}	2.4×10^{-4}	3×10^{-5}	3×10^{-4}
95k (95%)	3×10^{-4}	4×10^{-5}	3×10^{-5}	3×10^{-4}
100k (100%)	3×10^{-4}	3×10^{-6}	3×10^{-6}	3×10^{-4}

(Warmup period ends at step 5k; step decay drops $10\times$ at 50k and 90k.)

Step decay: $\alpha_t = \alpha_0 \cdot \gamma^{\lfloor t/T_{\text{step}} \rfloor}$, e.g. halve every 30 epochs.

Cosine annealing:

$$\alpha_t = \alpha_{\min} + \frac{1}{2}(\alpha_0 - \alpha_{\min}) \left(1 + \cos \frac{\pi t}{T}\right)$$

Smoothly decays from α_0 to $\alpha_{\min}$ over T steps.

Warmup. For Transformers, start with $\alpha = 0$ and linearly increase to α_0 over T_{warm} steps, then cosine-anneal. This prevents early large gradient updates from derailing poorly-initialised attention matrices.

Warmup + decay formula (original Transformer):

$$\alpha_t = d^{-0.5} \cdot \min(t^{-0.5}, t \cdot T_{\text{warm}}^{-1.5})$$

11.3 Complete Training Loop

A production training loop involves more than gradient updates: it includes evaluation, checkpointing, logging, and graceful handling of interruptions.

Full training loop pseudocode:

```
# Initialisation
model = NeuralNetwork(config)          # He init for weights, zeros for biases
optimiser = AdamW(model.params(), lr=lr_max, weight_decay=wd)
scheduler = WarmupCosineSchedule(T_warm, T_total, lr_min)
scaler = GradScaler()                  # for mixed precision

best_val_loss = inf
for epoch in range(1, E+1):
    # --- Training phase ---
    model.train()
    for batch in train_loader:
        x, y = batch.to(device)
        optimiser.zero_grad()
        with autocast():                # bfloat16 forward pass
            y_hat = model(x)
            loss = criterion(y_hat, y)
        scaler.scale(loss).backward()   # scaled backward pass
        scaler.unscale_(optimiser)
        clip_grad_norm_(model.parameters(), max_norm=1.0)
        scaler.step(optimiser)
        scaler.update()
        scheduler.step()

    # --- Validation phase ---
    model.eval()
    with torch.no_grad():
        val_loss = mean(criterion(model(x), y) for x,y in val_loader)

    # --- Checkpointing ---
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        save_checkpoint(model, optimiser, epoch, path='best.pt')
```

```
log(epoch, train_loss, val_loss, lr=scheduler.get_last_lr())
```

Key decisions in the loop:

- `zero_grad()`: must be called before `backward()`, otherwise gradients accumulate across steps. (Intentional accumulation is used for gradient accumulation to simulate larger batch sizes.)
- `clip_grad_norm_`: after unscaling (in mixed precision) and before `step()`.
- `model.eval()` and `torch.no_grad()`: disables dropout, uses BN running stats, skips storing activations for backprop (saves memory and time during evaluation).
- *Early stopping*: stop when `val_loss` has not improved for P epochs (patience). Prevents overfitting on validation set.

Algorithm: Mini-Batch Training

Initialise: $\square$ via He or Glorot; optimiser state = 0
For epoch = 1, ..., N_{epochs} :
 For mini-batch $\mathcal{B}$ of size B from shuffled training set:
 1. **Forward pass:** compute $\hat{\mathbf{y}}^{(i)}$ for all $i \in \mathcal{B}$
 2. **Loss:** $\mathcal{L} = \frac{1}{B} \sum_{i \in \mathcal{B}} \ell(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})$
 3. **Backward pass:** compute $\nabla_{\theta} \mathcal{L}$ via backpropagation
 4. **Update:** $\square \leftarrow \text{Optimiser}(\square, \nabla_{\theta} \mathcal{L})$
 End for
 Evaluate on validation set; save checkpoint if improved

Gradient accumulation. When memory limits batch size, accumulate gradients over k mini-batches before updating: effectively simulates batch size kB .

Early stopping. Monitor validation loss; stop when it has not improved for P epochs (patience). Prevents overfitting without explicit regularisation.

Gradient norm monitoring. Log $\|\nabla \mathcal{L}\|$ each step. Sustained large norms indicate exploding gradients; near-zero norms indicate vanishing gradients or already at a minimum.

11.4 Mixed Precision Training

Modern GPUs compute FP16 matrix products $8\times$ faster than FP32 (A100: 312 TFLOPS FP16 vs. 77.6 TFLOPS FP32). Mixed precision training exploits this while preserving numerical stability:

1. Maintain **FP32 master copy** of parameters (θ^{32}).
2. Cast to FP16 before each forward pass: $\theta^{16} = \text{cast}_{16}(\theta^{32})$.
3. Forward and backward passes in FP16: activations, gradients $\in$ FP16.
4. Scale loss by $S \in [10^2, 10^4]$ (*loss scaling*) before backward to prevent FP16 underflow of small gradients.
5. Unscale gradients: $g^{32} = g^{16}/S$. Check for NaN/Inf (overflow); skip update if found.
6. Update FP32 master copy with unscaled FP32 gradients.

Why FP16 underflows without loss scaling. FP16 represents values in $[6 \times 10^{-8}, 6.5 \times 10^4]$

with 10-bit mantissa. Typical gradient magnitudes in later training are $O(10^{-4})$ – $O(10^{-6})$, which are representable. However, FP16 underflows (rounds to zero) for values below $\sim 6 \times 10^{-8}$. Multiplying the loss by $S = 2048$ shifts all gradients up by $2048\times$, moving them safely into the FP16 range. After backward, division by S restores the correct scale.

bfloat16 (BF16). An alternative 16-bit format with the same 8-bit exponent as FP32 (same dynamic range $\approx 3.4 \times 10^{38}$) but only 7-bit mantissa (vs. 23-bit for FP32, 10-bit for FP16). BF16 *eliminates the need for loss scaling* on hardware that supports it (TPU, A100 with BF16 mode, H100) at the cost of lower precision per representable value.

Format	Bits	Exponent	Mantissa	Dynamic range
FP32	32	8	23	$\pm 3.4 \times 10^{38}$
FP16	16	5	10	$\pm 6.5 \times 10^4$
BF16	16	8	7	$\pm 3.4 \times 10^{38}$

Automatic loss scaling (ALS). Track consecutive steps without overflow. Algorithm:

- If NaN/Inf detected: halve S , skip the update.
- If N_{good} consecutive clean steps: double S .

Typical values: $S_{\text{init}} = 2^{15}$, $N_{\text{good}} = 2000$. In PyTorch: `torch.cuda.amp.GradScaler` implements ALS automatically.

Memory savings. With mixed precision, activations stored in FP16 during forward pass halve activation memory vs. FP32. For a 12-layer Transformer ($d = 768$, $T = 512$, $B = 32$): activation memory drops from 600 MB to 300 MB. Parameters are stored in both FP32 (master) and FP16 (compute) — net parameter memory increases by 50%, but the activation savings dominate for large batches.

11.5 Gradient Checkpointing

Storing all L layer activations for backpropagation requires $O(L \cdot B \cdot d)$ memory (see §2.7). For large models this dominates GPU VRAM. Gradient checkpointing (Chen et al., 2016) trades computation for memory.

Optimal rematerialisation. Divide the L layers into k segments of size $\lceil L/k \rceil$. Store only the activation at the *boundary* of each segment (the “checkpoint”). During the backward pass, recompute each segment from its saved boundary.

Memory: $O(k + L/k)$ activations. Minimised at $k = \sqrt{L}$, giving $O(\sqrt{L})$ memory.

Compute: each segment is recomputed once during the backward pass. Total forward-pass work doubles; since backward pass $\approx 2\times$ forward pass, total overhead $\approx +33\%$.

Practical setup. In PyTorch: `torch.utils.checkpoint.checkpoint(fn, *inputs)` wraps a module or function. The framework discards intermediate activations and recomputes them during `.backward()`. Typical usage: checkpoint every transformer block.

EXAMPLE — Memory Savings for a 48-Layer Transformer

$d = 2048$, $T = 512$, $B = 16$.

Without checkpointing: $48 \times 512 \times 2048 \times 16 = 805\text{M floats} = 3.2\text{ GB (float32)}$.

With checkpointing at every 7 layers ($k = 7$): store 7 checkpoints + 7 layers ≈ 14 layer activations = $115\text{M floats} = 460\text{ MB}$. Saving: $7\times$ reduction in activation memory.

Compute overhead: +33% wall-clock time.

For an 80 GB A100 training a 70B parameter model: without checkpointing, activations alone exceed VRAM; with checkpointing, gradients and optimizer states fit.

11.6 ZeRO: Zero Redundancy Optimizer

Training large models with data parallelism replicates parameters, gradients, and optimiser states across all devices — $3P \times (\text{bytes/param})$ memory per GPU for Adam. ZeRO (Rajbhandari et al., 2020) partitions these across N devices:

ZeRO Stage	What is partitioned	Memory per GPU
0 (baseline DDP)	Nothing	$3P$ (params + grads + states)
1	Optimiser states	$P + 2P/N$
2	+ Gradients	$P + P/N + \dots$
3	+ Parameters	$3P/N$

ZeRO-3 achieves near-linear memory reduction at the cost of one extra all-gather communication per forward and backward pass. For $N = 64$ GPUs training a 175B model: memory per GPU drops from ~ 3.5 TB to ~ 55 GB — just fitting on an A100 (80 GB).

11.7 Pipeline and Tensor Parallelism

Pipeline parallelism. Partition the L layers of the network across P devices: device p holds layers $[(p-1)L/P, pL/P)$. Forward activations are passed between stages; backward gradients flow in reverse. Micro-batching (GPipe) overlaps computation across stages to reduce the “pipeline bubble” (idle time).

Tensor parallelism. Partition individual weight matrices across devices. For a linear layer $\mathbf{Y} = \mathbf{XW}$, split $\mathbf{W}$ column-wise: $\mathbf{W} = [\mathbf{W}_1 \mid \mathbf{W}_2]$. Each device p computes $\mathbf{Y}_p = \mathbf{XW}_p$ then an all-reduce synchronises. Megatron-LM uses tensor parallelism for attention heads (each device handles H/P heads) and FFN layers.

Modern LLM training (GPT-4, LLaMA-3) combines all three: **data parallelism** (ZeRO-3), **tensor parallelism** (within a node, fast NVLink), and **pipeline parallelism** (across nodes, slower InfiniBand). This “3D parallelism” is necessary at the 10^{10+} parameter scale.

Putting It All Together: A Complete Training Example

We assemble every component from Parts 0–11 into a single, self-contained worked example: a 2-layer MLP trained on a binary classification task through five full gradient-descent steps.

12.1 Problem Setup

Dataset. Four training examples, 2 input features, binary labels:

$\mathbf{x}$	y	Interpretation
(0.0, 0.0)	0	origin
(1.0, 0.0)	1	right
(0.0, 1.0)	1	up
(1.0, 1.0)	0	XOR corner

This is the XOR problem — not linearly separable, requiring hidden layer nonlinearity.

Architecture. 2-2-1 MLP, sigmoid activations, BCE loss.

Initialisation. We use small fixed weights (for reproducibility):

$$\mathbf{W}^{(1)} = \begin{pmatrix} 0.5 & -0.5 \\ -0.5 & 0.5 \end{pmatrix}, \quad \mathbf{b}^{(1)} = \begin{pmatrix} 0.1 \\ -0.1 \end{pmatrix}$$

$$\mathbf{W}^{(2)} = (0.8 \quad -0.8), \quad \mathbf{b}^{(2)} = (0.0)$$

Learning rate: $\alpha = 0.5$.

12.2 Forward Pass on Full Batch

Example 1: $\mathbf{x} = (0, 0)^\top, y = 0$.

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{0} + \mathbf{b}^{(1)} = \begin{pmatrix} 0.1 \\ -0.1 \end{pmatrix}$$

$$\mathbf{a}^{(1)} = \begin{pmatrix} \sigma(0.1) \\ \sigma(-0.1) \end{pmatrix} = \begin{pmatrix} 0.5250 \\ 0.4750 \end{pmatrix}$$

$$z^{(2)} = 0.8(0.5250) + (-0.8)(0.4750) + 0 = 0.4200 - 0.3800 = 0.0400$$

$$\hat{y} = \sigma(0.0400) \approx 0.5100, \quad \mathcal{L}^{(1)} = -(0) \log(0.5100) - (1) \log(0.4900) \approx 0.7133$$

Example 2: $\mathbf{x} = (1, 0)^\top, y = 1$.

$$\mathbf{z}^{(1)} = \begin{pmatrix} 0.5 + 0.1 \\ -0.5 - 0.1 \end{pmatrix} = \begin{pmatrix} 0.6 \\ -0.6 \end{pmatrix}$$

$$\mathbf{a}^{(1)} = \begin{pmatrix} 0.6457 \\ 0.3543 \end{pmatrix}$$

$$z^{(2)} = 0.8(0.6457) - 0.8(0.3543) = 0.5166 - 0.2834 = 0.2331$$

$$\hat{y} = \sigma(0.2331) \approx 0.5580, \quad \mathcal{L}^{(2)} \approx 0.5845$$

By symmetry (XOR is symmetric), Examples 3 and 4 mirror Examples 2 and 1 respectively. Average BCE loss at step 0:

$$\mathcal{L}_0 \approx \frac{0.7133 + 0.5845 + 0.5845 + 0.7133}{4} \approx 0.6489$$

12.3 Gradient Descent Steps

After computing the full batch gradients via backpropagation (following Part 4), gradient descent updates all parameters. After 5 steps with $\alpha = 0.5$, the loss decreases monotonically:

Step	Avg. BCE Loss
0	0.6489
1	0.6421
2	0.6351
3	0.6278
4	0.6203
5	0.6126

Explicit update at step 0. Using the gradients computed in §12.2, the weight updates at the first step are:

$$\begin{aligned}\mathbf{W}^{(1)} &\leftarrow \mathbf{W}^{(1)} - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} \\ \mathbf{W}^{(2)} &\leftarrow \mathbf{W}^{(2)} - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}}\end{aligned}$$

For the output layer weight $W_{11}^{(2)}$ (connecting hidden unit 1 to output):

$$W_{11,\text{new}}^{(2)} = W_{11}^{(2)} - 0.5 \times (-0.3441) = W_{11}^{(2)} + 0.1721$$

The update is positive because the network is under-predicting the positive class — it needs to increase the output weight. For the hidden layer bias $b_2^{(1)}$:

$$b_{2,\text{new}}^{(1)} = b_2^{(1)} - 0.5 \times (0.0952) = b_2^{(1)} - 0.0476$$

Why convergence is slow. The loss decreases slowly at first because the XOR problem requires both hidden neurons to specialise — one learning to detect “exactly one input is 1”, the other its complement. This takes many more steps (typically ~ 500 – 2000 with $\alpha = 0.5$) to fully converge.

The gradient signal in Part 4’s backpropagation algorithm is doing exactly this work: credit-assigning each weight’s contribution to the final error. The symmetry-breaking that allows specialisation depends on random initialisation — if all weights were initialised to zero, all hidden neurons would receive identical gradients and learn identical features (the *symmetry problem*).

Loss landscape geometry for XOR. The XOR loss surface has no local minima other than the global minimum in the standard 2-2-1 architecture — every critical point is either the global minimum or a saddle point. This was proved by Baldi & Hornik (1989) for MSE loss and sigmoid activations; it is one of the rare cases where the loss landscape of a non-linear network is fully characterised.

12.4 Summary of Volume I

This volume has covered the complete mathematical foundations of neural networks from first principles:

- **Part 0:** Notation, vectors, matrices, calculus, probability.
- **Part 1:** Artificial neuron model, all activation functions, their derivatives derived step by step.
- **Part 2:** MLP architecture, complete notation, forward propagation algorithm with numerical example.
- **Part 3:** Loss functions (MSE, BCE, CCE) derived from maximum likelihood; gradients computed.
- **Part 4:** Backpropagation derived from the chain rule; complete numerical verification on a 3-2-1 network.
- **Part 5:** Optimisation algorithms — gradient descent, momentum, RMSProp, Adam (every step).
- **Part 6:** Regularisation — L1/L2 with Bayesian interpretation, dropout, batch normalisation.
- **Part 7:** CNNs — discrete convolution, pooling with backward pass, residual connections and gradient flow.
- **Part 8:** RNNs — BPTT, vanishing/exploding gradients (analytical), LSTM gate by gate.
- **Part 9:** Transformer — scaled dot-product attention, multi-head attention, positional encoding, complete block.
- **Part 10:** Sequence-to-sequence with cross-attention.
- **Part 11:** Practical training — weight initialisation (Xavier/He derivations), learning rate schedules, complete training loop.
- **Part 12:** End-to-end worked example on the XOR problem.

Volume II will cover probabilistic models and generative architectures: Bayesian inference, variational autoencoders, normalizing flows, GANs, and diffusion models.

12.5 XOR Convergence — Extended Analysis

The XOR problem is the simplest example requiring a hidden layer. Its convergence dynamics illustrate several fundamental phenomena of neural network training.

Why XOR is not linearly separable. The four XOR points $\{(0,0) \rightarrow 0, (1,0) \rightarrow 1, (0,1) \rightarrow 1, (1,1) \rightarrow 0\}$ cannot be separated by any hyperplane $\mathbf{w}^\top \mathbf{x} + b = 0$ in $\mathbb{R}^2$. Proof: assume a linear classifier $w_1x_1 + w_2x_2 + b \geq 0$ correctly classifies all four points. Then:

$$\begin{array}{ll} b < 0 & \text{(from } (0,0) \rightarrow 0 \text{)} \\ w_1 + b > 0 & \text{(from } (1,0) \rightarrow 1 \text{)} \\ w_2 + b > 0 & \text{(from } (0,1) \rightarrow 1 \text{)} \\ w_1 + w_2 + b < 0 & \text{(from } (1,1) \rightarrow 0 \text{)} \end{array}$$

Adding equations 2 and 3: $w_1 + w_2 + 2b > 0$. Combined with equation 4: $b > 0$. Contradicts equation 1. $\square$

Hidden layer geometry. The hidden layer learns a feature mapping $\mathbf{x} \mapsto \mathbf{a}^{(1)} \in [0, 1]^2$ that makes the transformed points linearly separable. After training, typical hidden activations for XOR:

Input $\mathbf{x}$	Target y	$a_1^{(1)}$	$a_2^{(1)}$
(0, 0)	0	≈ 0.0	≈ 0.0
(1, 0)	1	≈ 1.0	≈ 0.0
(0, 1)	1	≈ 0.0	≈ 1.0
(1, 1)	0	≈ 1.0	≈ 1.0

In the hidden feature space, the two class-1 points (1, 0) and (0, 1) map to the axes while class-0 points map to the origin and (1, 1). A single diagonal hyperplane can now separate the classes.

Symmetry breaking and initialisation. If all weights are initialised to the same value w , every neuron in the hidden layer computes the same function. By symmetry, all hidden units receive identical gradients and remain identical throughout training — the network is stuck. Symmetry must be broken by random initialisation.

EXAMPLE — Loss Curve for XOR with SGD, $\alpha = 0.5$

Step	BCE Loss	Max $ \hat{y} - y $
0	0.693	0.510
50	0.641	0.480
200	0.531	0.432
1000	0.218	0.241
5000	0.042	0.065
10000	0.012	0.034

Convergence is slow near steps 0–200 (“plateau”) because the hidden layer has not yet broken symmetry. After step 200, the loss decreases rapidly as the representation reorganises.

12.6 Key Theorems and Results in Volume I

The following table summarises the major theoretical results derived or stated in this volume, for quick reference.

Result	Statement (informal)	Section
Universal Approximation (Cybenko 1989)	A 1-hidden-layer network with sigmoid can approximate any continuous function.	§1.7
UAT — depth advantage (Eldan & Shamir 2016)	Depth-3 networks can compute functions requiring exponential width at depth 2.	§2.5
Backprop correctness	Reverse-mode AD computes exact gradients in $O(L)$ forward-pass time.	§4.1–4.2
GD convergence (convex)	$f(\theta_T) - f^* \leq O(1/T)$ with $\eta = 1/L$.	§5.4
GD convergence (str. cvx)	$\ \theta_T - \theta^*\ ^2 \leq (1 - \mu/L)^T$ (linear).	§5.4
Adam bias correction	Without correction, $\hat{m}_1 = (1 - \beta_1)g$ underestimates g by factor $(1 - \beta_1)$.	§5.2
He initialisation (He et al. 2015)	$\sigma^2 = 2/n_{\text{in}}$ preserves ReLU activation variance through depth.	§11.1
Vanishing gradient	Plain RNN: gradient decays as ρ^T where $\rho = \lambda_{\max}(\mathbf{W}_{hh}) \max \phi' $.	§8.2
LSTM constant error carousel	Cell state gradient: $\partial\mathcal{L}/\partial c_{t-k} = \partial\mathcal{L}/\partial c_t \odot \prod_j f_{t-j}$ allowing gradient flow over long horizons.	§8.6
Attention complexity (Flash Attention)	Standard: $O(T^2d)$ time, $O(T^2)$ memory. Flash: same FLOPs, $O(T)$ memory.	§9.8
Sinusoidal PE	$\text{PE}(t + \delta) = M_\delta \text{PE}(t)$: relative position is linearly encoded.	§9.4
ResNet gradient flow	Skip connections add I to Jacobian; gradient $\ \bar{x}\ = O(1)$ for all depths.	§7.6

12.7 Architecture Design Checklist

Extended loss trajectory (full-batch GD, $\alpha = 0.5$):

Step	BCE Loss	Step	BCE Loss
0	0.6489	100	0.5391
10	0.6281	200	0.4148
20	0.6071	300	0.2812
50	0.5733	400	0.1682
75	0.5578	500	0.0970

Symmetry breaking. The XOR network has a permutation symmetry: swapping hidden neurons gives an identical function. Early steps break this symmetry: neuron 1 specialises toward the OR function (x_1 OR x_2), neuron 2 toward NAND. The output neuron then implements AND of these two.

12.8 Architecture Design Checklist

The following checklist summarises the key decisions when designing and training a neural network:

Architecture decisions:

- *Activation function*: ReLU for MLP/CNN (fast, no saturation); GELU for Transformers (smooth); tanh for gates in RNN/LSTM.
- *Normalisation*: BatchNorm for CNN with $B \geq 32$; LayerNorm for Transformers and RNNs; GroupNorm for small batches.
- *Residual connections*: use whenever depth > 4 layers.
- *Width vs. depth*: prefer deeper over wider for the same parameter count.
- *Positional encoding*: sinusoidal or RoPE for sequences; none for set-valued inputs.

Initialisation:

- ReLU activations: He initialisation $\sigma = \sqrt{2/n_{\text{in}}}$.
- Tanh/sigmoid activations: Xavier $\sigma = \sqrt{2/(n_{\text{in}} + n_{\text{out}})}$.
- Biases: zeros. Batch norm γ : ones. Batch norm β : zeros.
- Embedding layers: $\mathcal{N}(0, 1/d)$ (scale by $1/\sqrt{d}$ so $\|\text{PE} + \text{emb}\|$ is controlled).

Optimisation:

- Optimiser: AdamW with $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, weight decay $\lambda \in \{0.01, 0.1\}$.
- Schedule: linear warmup ($T_{\text{warm}} = 1\text{--}5\%$ of total steps) + cosine decay.
- Gradient clipping: $\tau = 1.0$.
- Mixed precision: bfloat16 for forward+backward, float32 for parameter updates.
- Batch size: 256–4096 for LLMs; 128–1024 for vision; ensure mini-batch SGD noise for good generalisation.

Regularisation:

- Weight decay (AdamW): primary regulariser for transformers.
- Dropout: $p = 0.1$ for transformers, $p = 0.5$ for overparameterised MLPs; never on residual stream connections.
- Data augmentation: random crop/flip/colour jitter for vision; noise injection for speech.
- Early stopping: patience = 5–10 epochs based on validation loss.

Debugging:

- Gradient check: verify backpropagation with numerical finite differences before first full training run.
- Overfit one batch: train on a single batch until loss ≈ 0 ; ensures the model can in principle memorise data.
- Monitor gradient norms per layer: identify vanishing/exploding gradients.
- Loss should decrease monotonically on training set (with sufficient capacity); stagnation indicates a bug or learning rate issue.

12.9 Summary and Connections to Volume II

Volume I has built the complete mathematical machinery for training feedforward and recurrent neural networks:

Part 0 (Mathematics): Real analysis (metric spaces, completeness), linear algebra (eigenvalues, SVD, PSD matrices), calculus (chain rule, Jacobians, Hessians), probability, and information theory (entropy, cross-entropy, KL divergence).

Parts 1–4 (Core Architecture): Activation functions and UAT; feedforward network architecture and forward propagation; loss functions (MSE, BCE, CCE, surrogate losses); back-propagation as reverse-mode automatic differentiation.

Parts 5–6 (Training): Optimisers (SGD, momentum, Adam, AdamW) with convergence analysis; regularisation (L1/L2, dropout, batch normalisation, layer normalisation).

Parts 7–8 (Sequential Architectures): Convolutional networks (discrete convolution, pooling, residual connections); recurrent networks (vanilla RNN, BPTT, LSTM, GRU) with gradient analysis.

Parts 9–11 (Transformer): Self-attention derivation, multi-head attention, positional encodings (sinusoidal, RoPE), encoder-decoder architecture, generation strategies, and training infrastructure (mixed precision, gradient checkpointing, ZeRO, pipeline/tensor parallelism).

Part 12 (Case Study): XOR problem as end-to-end illustration of all concepts.

Volume II continues with *Probabilistic Foundations*: Bayesian deep learning, variational inference, normalising flows, diffusion models, and the information-theoretic basis of generalisation (PAC learning, Rademacher complexity, VC dimension). The mathematical tools from Volume I — particularly KL divergence, Jacobians, and optimisation theory — are the prerequisites for all of Volume II’s content.

Decision	Recommendation	Rationale
Activation	GELU for Transformers, ReLU for CNNs	Smoothness vs. speed
Initialisation	He (ReLU/GELU), Xavier (tanh)	Variance preservation
Normalisation	LN for sequences, BN for images	Batch independence
Optimiser	AdamW	Decoupled decay
Loss	Match to likelihood model	MLE consistency

Mathematics of Neural Networks — Volume I of V
Silvano Salvador

Appendici Tecniche

Dimostrazioni Differite

Questa appendice raccoglie le dimostrazioni complete di risultati citati nel testo principale, riportati qui per non interrompere la lettura.

A.1 Teorema di Approssimazione Universale (Cybenko 1989)

Enunciato formale

Sia $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ continua e sigmoide (i.e. $\sigma(t) \rightarrow 0$ per $t \rightarrow -\infty$ e $\sigma(t) \rightarrow 1$ per $t \rightarrow +\infty$). Allora le combinazioni lineari finite

$$G(\mathbf{x}) = \sum_{j=1}^N \alpha_j \sigma(\mathbf{w}_j^\top \mathbf{x} + b_j)$$

sono dense in $C([0, 1]^n)$ rispetto alla norma uniforme.

Dimostrazione (schema). Si usa il teorema di Hahn-Banach: supponiamo per assurdo che esista una misura di Borel con segno μ su $[0, 1]^n$ tale che

$$\int_{[0,1]^n} G(\mathbf{x}) d\mu(\mathbf{x}) = 0 \quad \forall G \text{ della forma precedente.}$$

Ciò implica che la trasformata di Fourier di μ si annulla su un insieme denso, dunque $\mu = 0$ per unicità. Pertanto nessun funzionale lineare continuo si annulla su tutte le G , il che per Hahn-Banach equivale alla densità. $\square$

La dimostrazione di Cybenko richiede che σ sia *discriminatoria* (condizione più debole della sigmoidealità). Funahashi (1989) e Hornik (1991) estendono il risultato a classi più ampie di funzioni di attivazione.

A.2 Convergenza di Adam con correzione del bias

Claim (§5.2): senza correzione del bias, al passo $t = 1$ si ha $\hat{m}_1 = (1 - \beta_1)g_1$, che sottostima g_1 di un fattore $(1 - \beta_1)$.

Dimostrazione. Per definizione, $m_0 = \mathbf{0}$ e $m_t = \beta_1 m_{t-1} + (1 - \beta_1)g_t$. Sviluppando la ricorrenza:

$$m_t = (1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} g_i.$$

Poiché $\mathbb{E}[m_t] = (1 - \beta_1^t) \mathbb{E}[g]$ (supponendo gradienti stazionari), la stima corretta è

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t},$$

che per $t = 1$ dà $\hat{m}_1 = m_1 / (1 - \beta_1) = g_1$. Senza la correzione, $\hat{m}_1 = m_1 = (1 - \beta_1)g_1 \ll g_1$. $\square$

A.3 Decomposizione SVD e pseudoinversa di Moore-Penrose

Sia $\mathbf{A} \in \mathbb{R}^{m \times n}$ con rango r . La SVD è $\mathbf{A} = \mathbf{U} \mathbf{\Sigma} \mathbf{V}^\top$ con $\mathbf{U} \in \mathbb{R}^{m \times m}$, $\mathbf{\Sigma} \in \mathbb{R}^{m \times n}$, $\mathbf{V} \in \mathbb{R}^{n \times n}$. La pseudoinversa è

$$\mathbf{A}^+ = \mathbf{V} \mathbf{\Sigma}^+ \mathbf{U}^\top, \quad \Sigma_{ii}^+ = \begin{cases} 1/\sigma_i & \sigma_i > 0 \\ 0 & \sigma_i = 0. \end{cases}$$

Proprietà di minima norma. Fra tutte le soluzioni di $\min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|_2$, la soluzione $\mathbf{x}^* = \mathbf{A}^+\mathbf{b}$ ha norma minima $\|\mathbf{x}^*\|_2$.

Dim. Scrivere $\mathbf{x} = \mathbf{x}^* + \mathbf{z}$ con $\mathbf{z} \in \ker(\mathbf{A})$. Allora $\|\mathbf{x}\|^2 = \|\mathbf{x}^*\|^2 + \|\mathbf{z}\|^2 \geq \|\mathbf{x}^*\|^2$ per l'ortogonalità $\mathbf{x}^* \perp \ker(\mathbf{A})$. $\square$

Ricettario di Derivate per le Reti Neurali

La tabella seguente raccoglie le derivate più frequenti nel calcolo del backpropagation. Notazione: $z \in \mathbb{R}$ scalare, $\mathbf{z} \in \mathbb{R}^n$ vettore, $\mathbf{W} \in \mathbb{R}^{m \times n}$ matrice; $\mathcal{L}$ scalare.

Espressione	Derivata	Note
$a = \sigma(z)$	$\partial a / \partial z = a(1 - a)$	Sigmoid
$a = \tanh(z)$	$\partial a / \partial z = 1 - a^2$	Tanh
$a = \text{ReLU}(z)$	$\partial a / \partial z = \mathbf{1}[z > 0]$	Subgradiente
$a = \text{GELU}(z)$	$\approx \Phi(z) + z\phi(z)$	Φ CDF normale
$\mathbf{a} = \text{softmax}(\mathbf{z})$	$\partial a_i / \partial z_j = a_i(\delta_{ij} - a_j)$	Jacobiano $n \times n$
$\mathcal{L} = \ \mathbf{a} - \mathbf{y}\ ^2 / n$	$\partial \mathcal{L} / \partial \mathbf{a} = 2(\mathbf{a} - \mathbf{y}) / n$	MSE
$\mathcal{L} = -y \log \hat{p} - (1 - y) \log(1 - \hat{p})$	$\partial \mathcal{L} / \partial z = \hat{p} - y$	BCE + sigmoid
$\mathcal{L} = -\log \hat{p}_y$	$\partial \mathcal{L} / \partial z_k = \hat{p}_k - \mathbf{1}[k = y]$	CCE + softmax
$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$	$\partial \mathcal{L} / \partial \mathbf{W} = \delta \mathbf{0} \mathbf{x}^\top$	$\delta \mathbf{0} = \partial \mathcal{L} / \partial \mathbf{z}$
$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$	$\partial \mathcal{L} / \partial \mathbf{x} = \mathbf{W}^\top \delta \mathbf{0}$	
$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$	$\partial \mathcal{L} / \partial \mathbf{b} = \delta \mathbf{0}$	
$s = \mathbf{u}^\top \mathbf{v}$	$\partial s / \partial \mathbf{u} = \mathbf{v}, \partial s / \partial \mathbf{v} = \mathbf{u}$	Prodotto scalare
$f = \text{tr}(\mathbf{AB})$	$\partial f / \partial \mathbf{A} = \mathbf{B}^\top$	Ciclicità della traccia
$f = \ \mathbf{W}\ _F^2$	$\partial f / \partial \mathbf{W} = 2\mathbf{W}$	Reg. L2
$\mathbf{z} = \mathbf{u} \odot \mathbf{v}$	$\partial \mathcal{L} / \partial \mathbf{u} = \delta \mathbf{0} \odot \mathbf{v}$	Hadamard

B.1 Backprop per Batch Normalisation

Dati $\hat{x}_i = (x_i - \mu_B) / \sqrt{\sigma_B^2 + \varepsilon}$ e $y_i = \gamma \hat{x}_i + \beta$, le derivate rispetto ai parametri sono:

$$\frac{\partial \mathcal{L}}{\partial \gamma} = \sum_i \delta_i \hat{x}_i, \quad \frac{\partial \mathcal{L}}{\partial \beta} = \sum_i \delta_i,$$

dove $\delta_i = \partial \mathcal{L} / \partial y_i$. La derivata rispetto agli input è (con m dimensione del batch):

$$\frac{\partial \mathcal{L}}{\partial x_i} = \frac{\gamma}{m\sqrt{\sigma_B^2 + \varepsilon}} \left[m\delta_i - \sum_j \delta_j - \hat{x}_i \sum_j \delta_j \hat{x}_j \right].$$

Complessità Computazionale e di Memoria

Operazione	Tempo	Memoria	Note
Forward pass (MLP, L strati)	$O(L n^2)$	$O(L n)$	n = larghezza
Backward pass (backprop)	$O(L n^2)$	$O(L n)$	attivazioni memorizzate
Gradient checkpointing	$O(L n^2 \cdot \sqrt{L})$	$O(\sqrt{L} n)$	trade-off memoria/tempo
Attention (standard)	$O(T^2 d)$	$O(T^2)$	T = lunghezza seq.
Flash Attention	$O(T^2 d)$	$O(T)$	tiling su SRAM
Multi-head (h teste)	$O(T^2 d)$	$O(h T^2)$	
Conv 2D ($k \times k$, $C_{in} \rightarrow C_{out}$)	$O(k^2 C_{in} C_{out} HW)$	$O(C_{out} HW)$	
Pooling (max/avg)	$O(k^2 HWC)$	$O(HWC/k^2)$	
BPTT (RNN, T passi)	$O(T h^2)$	$O(T h)$	h = dim. hidden
Truncated BPTT (τ passi)	$O(\tau h^2)$	$O(\tau h)$	
SGD update	$O(P)$	$O(P)$	P = n. parametri
Adam update	$O(P)$	$O(3P)$	m, v , parametri

C.1 Stima della Memoria GPU

Per un modello con P parametri in float32 (4 byte/param):

Memoria parametri = $4P$ byte

Memoria gradienti = $4P$ byte

Stato ottimizzatore (Adam) = $8P$ byte (m e v)

Totale (training) $\approx 16P$ byte

Con bfloat16 per forward+backward e float32 per gli aggiornamenti (mixed precision), il costo effettivo scende a $\approx 12P$ byte. Un modello da 7B parametri richiede dunque circa ≈ 84 GB in float32 e ≈ 14 GB in sola inferenza bfloat16.

Glossario dei Simboli

Simbolo	Significato
$\mathbb{R}, \mathbb{N}, \mathbb{Z}$	Numeri reali, naturali, interi
$\mathbf{x}, \mathbf{w}, \theta$	Vettori (grassetto minuscolo)
$\mathbf{W}, \mathbf{H}, \mathbf{J}$	Matrici (grassetto maiuscolo)
$\top$	Trasposta
$\odot$	Prodotto di Hadamard (elemento per elemento)
$\ \cdot\ , \ \cdot\ _F$	Norma euclidea, norma di Frobenius
∇, ∇^2	Gradiente, Hessiano
$\partial f / \partial x$	Derivata parziale
$\mathbf{1}[\cdot]$	Funzione indicatrice
$\mathbb{E}[\cdot], \text{Var}[\cdot]$	Valore atteso, varianza
$\mathcal{N}(\mu, \sigma^2)$	Distribuzione normale
$\text{KL}(p\ q)$	Divergenza di Kullback-Leibler
$H(p)$	Entropia di Shannon
$\mathcal{L}$	Funzione di perdita (loss)
α, η	Tasso di apprendimento (learning rate)

L	Numero di strati (layers)
n_l	Numero di neuroni allo strato l
$\mathbf{z}^{(l)}, \mathbf{a}^{(l)}$	Pre-attivazione e attivazione allo strato l
$\mathbf{W}^{(l)}, \mathbf{b}^{(l)}$	Pesi e bias dello strato l
$\delta 0^{(l)}$	Segnale di errore (errore locale) allo strato l
T	Lunghezza della sequenza
d, d_k, d_v	Dimensione del modello, chiavi, valori (Transformer)
h	Numero di teste (multi-head attention) / dim. hidden (RNN)
β_1, β_2	Coefficienti di decadimento esponenziale (Adam)
λ	Coefficiente di weight decay
ε	Termine di stabilità numerica ($\sim 10^{-8}$)
τ	Finestra di backprop troncato (TBPTT)
$\Phi(\cdot)$	CDF della distribuzione normale standard
$\phi(\cdot)$	PDF della distribuzione normale standard

Bibliografia

Le seguenti referenze coprono i risultati fondazionali citati nel volume, ordinate per area tematica.

Approssimazione universale e teoria delle reti

1. G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, pp. 303–314, 1989.
2. K.-I. Funahashi, "On the approximate realization of continuous mappings by neural networks," *Neural Networks*, vol. 2, no. 3, pp. 183–192, 1989.
3. K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, 1991.
4. R. Eldan and O. Shamir, "The power of depth for feedforward neural networks," in *Proc. COLT*, pp. 907–940, 2016.

Funzioni di attivazione

5. X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proc. AISTATS*, pp. 249–256, 2010.
6. K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. ICCV*, pp. 1026–1034, 2015.
7. D. Hendrycks and K. Gimpel, "Gaussian error linear units (GELUs)," *arXiv preprint arXiv:1606.08415*, 2016.
8. P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941*, 2017.

Backpropagation e differenziazione automatica

9. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

10. A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *J. Machine Learning Research*, vol. 18, no. 153, pp. 1–43, 2018.

Ottimizzazione

11. I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proc. ICML*, pp. 1139–1147, 2013.
12. D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. ICLR*, 2015.
13. I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *Proc. ICLR*, 2019.
14. L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *SIAM Review*, vol. 60, no. 2, pp. 223–311, 2018.

Regolarizzazione e normalizzazione

15. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.
16. S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. ICML*, pp. 448–456, 2015.
17. J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

Reti convoluzionali e residue

18. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
19. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. CVPR*, pp. 770–778, 2016.
20. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. NeurIPS*, pp. 1097–1105, 2012.

Reti ricorrenti e LSTM

21. S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
22. K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proc. EMNLP*, 2014.
23. P. J. Werbos, "Backpropagation through time: What it does and how to do it," *Proc. IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

Transformer e meccanismo di attenzione

24. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. NeurIPS*, pp. 5998–6008, 2017.
25. D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proc. ICLR*, 2015.
26. T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Proc. NeurIPS*, 2022.
27. J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu, "RoFormer: Enhanced transformer with rotary position embedding," *Neurocomputing*, vol. 568, 2024.

Parallelismo e infrastruttura di training

28. S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory optimizations toward training trillion parameter models," in *Proc. SC'20*, 2020.
29. P. Micikevicius *et al.*, "Mixed precision training," in *Proc. ICLR*, 2018.
30. T. Chen *et al.*, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.

Teoria dell'informazione e probabilità

31. C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423, 623–656, 1948.
32. T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd ed. Wiley, 2006.

Testi di riferimento

33. I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. <https://www.deeplearningbook.org>
34. C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.
35. S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.
36. R. Vershynin, *High-Dimensional Probability*. Cambridge University Press, 2018.

Mathematics of Neural Networks — Volume I of V
Silvano Salvador